# Chapter 3 Domain Engineering[15]

## 3.1 What Is Domain Engineering?

*Domain Engineering*

[15] This is a chapter from K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. thesis, Technische Universität Ilmenau, Germany, 1998. This material will be also published in the upcoming book K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, to appear in 1999.

Most software systems can be classified according to the business area and the kind of tasks they support, e.g. airline reservation systems, medical record systems, portfolio management systems, order processing systems, inventory management systems, etc. Similarly, we can also classify *parts* of software systems according to their functionality, e.g. database systems, synchronization packages, workflow systems, GUI libraries, numerical code libraries, etc. We refer to areas organized around classes of systems or parts of systems as *domains*.[16]

Obviously, specific systems or components within a domain share many characteristics since they also share many requirements. Therefore, an organization which has built a number of systems or components in a particular domain can take advantage of the acquired knowledge when building subsequent systems or components in the same domain. 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. *Domain Engineering* is a systematic approach to achieving this goal.

> *Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable workproducts), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems.*

*Domain Analysis, Domain Design, and Domain Implementation*

Domain Engineering encompasses three main process components[17] *Domain Analysis*, *Domain Design*, and *Domain Implementation*. The main purpose of each of these components is given in Table 3.

*Application Engineering*

The results of Domain Engineering are reused during *Application Engineering*, i.e. the process of building a particular system in the domain (see Figure 8).

| **Domain Engineering process component** | **Main purpose** |
|---|---|
| Domain Analysis | defining a set of reusable requirements for the systems in the domain |
| Domain Design | establishing a common architecture for the systems in the domain |
| Domain Implementation | implementing the reusable assets, e.g. reusable components, domain-specific languages, generators, and a reuse infrastructure |

**Table 3** *Three Main Process Components of Domain Engineering*

Table 3 makes the distinction between the conventional software engineering and Domain Engineering clear: while the conventional software engineering concentrates on satisfying the requirements for a *single* system, Domain Engineering concentrates on providing *reusable* solutions for *families* of systems. By putting the qualifier "domain" in front of analysis,

[16] We give a precise definition of a domain in Section 3.6.1.

[17] Most of the current Domain Engineering methods still refer to the process components as *phases*. Following a recent trend in the software development methods field, we do not refer to analysis, design, and implementation as *phases* since the term *phase* implies some rigid, waterfall-style succession of engineering steps. Modern process models, such as the Rational Objectory Process, consider analysis, design, and implementation as *process components*. These are independent of the time dimension, which is itself divided into phases (see Section 4.5.1). In this newer terminology, however, phases indicate the maturity of the project over time. *Important note*: In order to be consistent with the original literature, the descriptions of the existing Domain Engineering methods in Section 3.7 use the term *phase* in its older meaning (i.e. to denote process components).

design, and implementation, we emphasize exactly this *family orientation* of the Domain Engineering process components.

*Software system engineering methods*

Indeed, if you take a look at the intentions of most of the current software engineering methods (including object-oriented analysis and design methods), you will realize that these methods aim at the development of "*this specific* system for *this specific* customer and for *this specific* context." We refer to such methods *software system engineering methods.*

*Multi-system scope development*

Domain Engineering, on the other hand, aims at the development of reusable software, e.g. a generic system from which you can instantiate concrete systems or components to be reused in different systems. Thus, Domain Engineering has to take into account different sets of customers (including potential ones) and usage contexts. We say that Domain Engineering addresses *multi-system scope* development.

Domain Engineering can be applied to a variety of problems, such as development of domain-specific frameworks, component libraries, domain-specific languages, and generators. The Domain Analysis process subcomponent of Domain Engineering, in particular, can also be applied to non-software-system-specific domains. For example, it has been used to prepare surveys, e.g. a survey of Architecture Description Languages [Cle96, CK95].

At the beginning of this section, we said that there are domains of systems and domains of parts of systems (i.e. *subsystems*). The first kind of domains is referred to as *vertical domains* (e.g. domain of medical record systems, domain of portfolio management systems, etc.) and the second kind is referred to as *horizontal domains* (e.g. database systems, numerical code libraries, financial components library, etc.). The product of Domain Engineering applied to a vertical domain is reusable software which we can instantiate to yield any concrete system in the domain. For example, we could produce a *system framework* (i.e. reusable system architecture plus components) covering the scope of a entire vertical domain. On the other hand, applying Domain Engineering to a horizontal domain yields reusable subsystems, i.e. *components*. We will come back to the notion of vertical and horizontal domains in Section 3.6.2.

*Components and other reusable assets*

In our terminology, a component is a *reusable* piece of software which is used to build more complex software. However, as already indicated, components are not the only workproducts of Domain Engineering. Other workproducts include reusable requirements, analysis and design models, architectures, patterns, generators, domain-specific languages, frameworks, etc. In general, we refer to any reusable workproduct as a *reusable asset*.

## 3.2 Domain Engineering and Related Approaches

Domain Engineering addresses the following two aspects:

- *Engineering of reusable software*: Domain Engineering is used to produce reusable software.
- *Knowledge management*: Domain Engineering should not be a "one-shot" activity. Instead, it should be a continuous process whose main goal is to maintain and update the knowledge in the domain of interest based on experience, scope broadening, and new trends and insights (see [Sim91] and [Arr89]).

*Organizational Memory, Design Rationale, and Experience Factory*

Current Domain Engineering methods concentrate on the first aspect and do not support knowledge evolution. The knowledge management aspect is addressed more adequately in the work on *Organizational Memory* [Con97, Buc97], *Design Rationale* [MC96], and *Experience Factory* [BCR94]. Three approaches have much in common with Domain Engineering, although they all come from different directions and each of them has a different focus:

- Domain Engineering concentrates on delivering reusable software assets.
- Organizational Memory concentrates on providing a common medium and an organized storage for the informal communication among a group of designers.

- Design Rationale research is concerned with developing effective methods and representations for capturing, maintaining and reusing records of the issues and trade-offs considered by designers during design and the ultimate reasons for the design decisions they make.

- Experience Factory provides a means for documenting the experience collected during past projects. It primarily concentrates on conducting mostly quantitative measurements and the analysis of the results.

As the research in these four areas advances, the overlap between them becomes larger. We expect that future work on Domain Engineering will address the knowledge management aspect to a larger degree (e.g. [Bai97]). In this chapter, however, we exclusively focus on the "engineering reusable software" aspect of Domain Engineering.

## 3.3 Domain Analysis

The purpose of *Domain Analysis* is to

- select and define the domain of focus and

- collect relevant domain information and integrate it into a coherent *domain model*.

The sources of domain information include existing systems in the domain, domain experts, system handbooks, textbooks, prototyping, experiments, already known requirements on future systems, etc.

It is important to note that Domain Analysis does not only involve recording the existing domain knowledge. The systematic organization of the existing knowledge enables and encourages us to actually extend it in creative ways. Thus, Domain Analysis is a *creative* activity.

*Domain model: commonalities, variabilities, and dependencies*

A *domain model* is an explicit representation of the *common* and the *variable* properties of the systems in a domain and the *dependencies* between the variable properties. In general, a domain model consists of the following components:

*Domain Definition*

- *Domain definition*: A domain definition defines the scope of a domain and characterizes its contents by giving examples of systems in a domain, counterexamples (i.e. systems outside the domain), and generic rules of inclusion or exclusion (e.g. "Any system having the capability X belongs to the domain.").

*Domain Lexicon*

- *Domain lexicon*: A domain lexicon defines the domain vocabulary.

*Concept model*

- *Concept models*: Concept models describe the concepts in a domain expressed in some appropriate modeling formalism (e.g. object diagrams, interaction and state-transition diagrams, or entity-relationship and data-flow diagrams).

*Feature model*

- *Feature models*: Feature models define a set of reusable and configurable requirements for specifying the systems in a domain. Such requirements are generally referred to as *features*. A feature model prescribes which feature combinations are meaningful: It represents the configuration aspect of the reusable software. We discuss feature models in Chapter 5.4 in great detail.

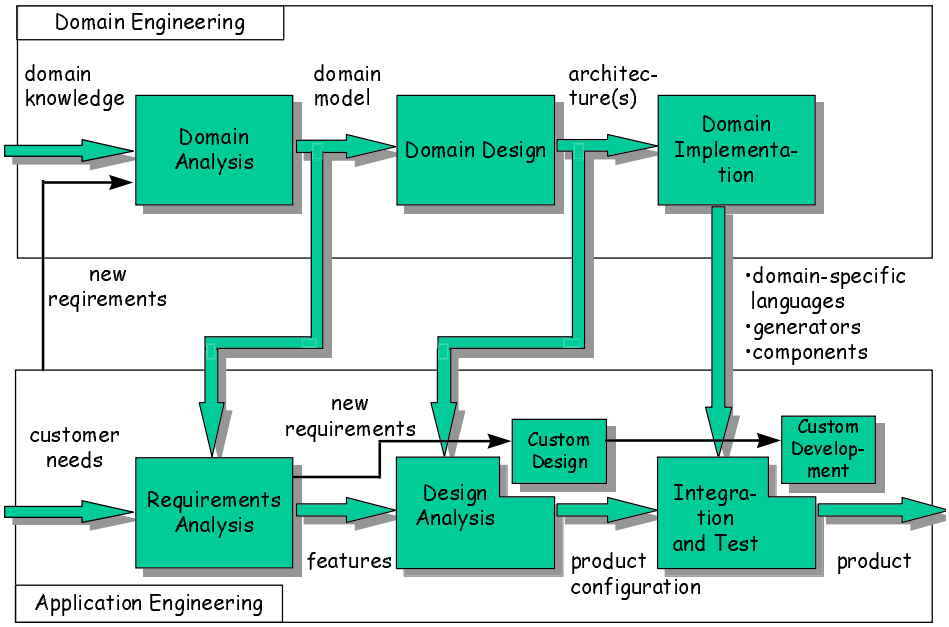



**Figure 8** *Software development based on Domain Engineering (adapted from [MBSE97])*

Domain Analysis generally involves the following activities:

*Domain planning, identification, and scoping*

- *Domain planning, identification, and scoping*: planning of the resources for performing domain analysis, identifying the domain of interest, and defining the scope of the domain;

*Domain modeling*

- *Domain modeling*: developing the domain model.

Table 4 gives you a more detailed list of Domain Analysis activities. This list was compiled by Arrango [Arr94] based on the study of eight different Domain Analysis methods.

| Domain Analysis major process components | Domain Analysis activities |
|---|---|
| *Domain characterization*<br><br>*(domain planning and scoping)* | *Select domain*<br>Perform business analysis and risk analysis in order to determine which domain meets the business objectives of the organization. |
| | *Domain description*<br>Define the boundary and the contents of the domain. |
| | *Data source identification*<br>Identify the sources of domain knowledge. |
| | *Inventory preparation*<br>Create inventory of data sources. |
| *Data collection*<br><br>*(domain modeling)* | *Abstract recovery*<br>Recover abstractions |
| | *Knowledge elicitation*<br>Elicit knowledge from experts |
| | *Literature review* |
| | *Analysis of context and scenarios* |
| *Data analysis*<br><br>*(domain modeling)* | *Identification of entities, operations, and relationships* |
| | *Modularization*<br>Use some appropriate modeling technique, e.g. object-oriented analysis or function and data decomposition. Identify design decisions. |
| | *Analysis of similarity*<br>Analyze similarities between entities, activities, events, relationships, structures, etc. |
| | *Analysis of variations*<br>Analyze variations between entities, activities, events, relationships, structures, etc. |
| | *Analysis of combinations*<br>Analyze combinations suggesting typical structural or behavioral patterns. |
| | *Trade-off analysis*<br>Analyze trade-offs that suggest possible decompositions of modules and architectures to satisfy incompatible sets of requirements found in the domain. |
| *Taxonomic classification*<br><br>*(domain modeling)* | *Clustering*<br>Cluster descriptions. |
| | *Abstraction*<br>Abstract descriptions. |
| | *Classification*<br>Classify descriptions. |
| | *Generalization*<br>Generalize descriptions. |
| | *Vocabulary construction* |
| *Evaluation* | Evaluate the domain model. |

**Table 4** *Common Domain Analysis process by Arrango [Arr94]*

## 3.4 Domain Design and Domain Implementation

*Software architecture*

The purpose of *Domain Design* is to develop an *architecture* for the systems in the domain. Shaw and Garlan define software architecture as follows [SG96]:

> *"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system design."*

Buschmann et al. offer another definition of software architecture [BMR+96]:

> *A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is the result of the software development activity.*

Just as the architecture of a building is usually represented using different views (e.g. static view, dynamic view, specification of materials, etc.), the adequate description of a software architecture also requires multiple views. For example, the 4+1 View Model of software architecture popularized by the Rational methodologist Philippe Kruchten consists of a logical view (class, interaction, collaboration, and state diagrams), a process view (process diagrams), a physical view (package diagrams), a deployment view (deployment diagrams), plus a use case model (see Figure 17).

The elements and their connection patterns in a software architecture are selected to satisfy the requirements on the system (or the systems) described by the architecture. When developing a software architecture, we have to consider not only functional requirements, but also nonfuctional requirements such as performance, robustness, failure tolerance, throughput, adaptability, extendibility, reusability, etc. Indeed, one of the purposes of software architecture is to be able to quickly tell how the software satisfies the requirements. Eriksson and Penker [EP98] say that "architecture should serve as a map for the developers, revealing how the system is constructed and where specific functions or concepts are located."

*Architectural patterns*

Certain recurring arrangements of elements have proven to be particularly useful in many designs. We refer to these arrangements as *architectural patterns*. Each architectural pattern aims at satisfying a different set of requirements. Buschman et al. have compiled a (partial) list of architectural patterns (see [BMR+96] for a detailed description of these patterns):

- *Layers pattern*: An arrangement into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

- *Pipes and filters pattern*: An arrangement that processes a stream of data, where a number of processing steps are encapsulated in filter components. Data is passed through pipes between adjacent filters, and the filters can be recombined to build related systems or system behavior.

- *Blackboard pattern*: An arrangement where several specialized subsystems assemble their knowledge to build a partial or approximate solution to a problem for which no deterministic solution strategy is known.

- *Broker pattern*: An arrangement where decoupled components interact by remote service invocations. A broker component is responsible for coordinating communication and for transmitting results and exceptions.

- *Model-view-controller pattern*: A decomposition of an interactive system into three components: A model containing the core functionality and data, one or more views

displaying information to the user, and one or more controllers that handle user input. A change-propagation mechanism ensures consistency between user interface and model.

- *Microkernel pattern*: An arrangement that separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

It is important to note that real architectures are usually based on more than one of these and other patterns at the same time. Different patterns may be applied in different parts, views, and at different levels of an architecture.

The architectural design of a system is a high-level design: it aims at coming up with a flexible structure which satisfies all important requirements and still leaves a large degree of freedom for the implementation. The architecture of a family of systems has to be even more flexible since it must cover different sets of requirements. In particular, it has to include an explicit representation of the variability (i.e. configurability) it covers so that concrete architectures can be configured based on specific sets of requirements. One way to capture this variability is to provide configuration languages for the configurable parts of the architecture. We will see a concrete example of a configuration language in Chapter 10.

A flexible architecture is the prerequisite for enabling the evolution of a system. As a rule, we use the most stable parts to form the "skeleton" and keep the rest flexible and easy to evolve. But even the skeleton has to be sometimes modified. Depending on the amount of flexibility an architecture provides, we distinguish between generic and highly flexible architectures [SCK+96]:

*Generic vs. highly flexible architectures*

- *Generic architecture*: A system architecture which generally has a fixed topology but supports component plug-and-play relative to a fixed or perhaps somewhat variable set of interfaces. We can think of a generic architecture as a frame with a number of sockets where we can plug in some alternative or extension components. The components have to clearly specify their interfaces, i.e. what they expect and what they provide.

- *Highly flexible architecture*: An architecture which supports structural variation in its topology, i.e. it can be configured to yield a particular generic architecture. The notion of a highly flexible architecture is necessary since a generic architecture might not be able to capture the structural variability in a domain of highly diverse systems. In other words, a flexible architecture componentizes even the "skeleton" and allows us to configure it and to evolve it over time.

Software architecture is a relatively young field with a very active research. You will find more information on this topic in [SG96, BMR+96, Arch, BCK98].

*Domain Implementation*

Domain Design is followed by Domain Implementation. During *Domain Implementation* we apply appropriate technologies to implement components, generators for automatic component assembly, reuse infrastructure (i.e. component retrieval, qualification, dissemination, etc.), and application production process.[18]

## 3.5 Application Engineering

*Application Engineering* is the process of building systems based on the results of Domain Engineering (see Figure 8). During the requirements analysis for a new systems, we take advantage of the existing domain model and select the requirements (*features*) from the domain model which match customer needs. Of course, new customer requirements not found in the domain model require custom development. Finally, we assemble the application from

[18] Some authors (e.g. [FPF96, p. 2]) divide Domain Engineering into only two parts, Domain Analysis and Domain Implementation, and regard the development of an architecture merely as an activity in the Domain Implementation.

the existing reusable components and the custom-developed components according to the reusable architecture, or, ideally, let a generator do this work.

## 3.6 Selected Domain Engineering Concepts

In the following sections, we discuss a number of basic concepts related to Domain Engineering: *domain*, *domain scope*, *relationships between domains*, *problem space* and *solution space*, and *specialized Domain Engineering methods*.

### 3.6.1 Domain

The American Heritage Dictionary of the English Language gives us a very general definition of a domain [Dict]:

> *"Domain: A sphere of activity, concern, or function; a field, e.g. the domain of history."*

According to this definition, we can view a domain as a body of knowledge organized around some focus, such as a certain professional activity.

Simos et al. note that the term *domain* is used in different disciplines and communities, such as linguistics, cultural research, artificial intelligence (AI), object-oriented technology (OO), and software reuse, in somewhat different meanings [SCK+96, p. 20]. They distinguish two general usage categories of this term:

1. *domain as the "real world"*;
2. *domain as a set of system*.

The notion of *domain as the "real world"* is used in the AI and knowledge-engineering communities. For example, the guidebook on Knowledge-Based Systems Analysis and Design Support (KADS), which is a prominent method for developing knowledge-based systems, gives the following definition [TH93, p. 495]:

> *"Domain: An area of or field of specialization where human expertise is used, and a Knowledge-Based System application is proposed to be used within it."*

Domain as the "real world" encapsulates the knowledge about the problem area (e.g. *accounts*, *customers*, *deposits and withdrawals*, etc., in a *bank accounting domain*), but not about the software from this problem area. This notion of domain as the "real world" is also used in object-oriented technology. For example, the UML (Unified Modeling Language) glossary defines domain as follows [UML97a]:

> *"Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area."*

In the context of software reuse and particularly in Domain Engineering, the term *domain* encompasses not only the "real world" knowledge but also the knowledge about how to build software systems in the domain of interest. Some early definitions even equate domain to a set of systems, e.g. [KCH+90, p. 2]:

> *"Domain: A set of current and future applications which share a set of common capabilities and data."*

or [Bai92, p. 1]:

> *"Domains are families of similar systems."*

This *domain as a set of systems* view is more appropriately interpreted as the assertion that a domain encompasses the knowledge used to build a family of software systems.

It is essential to realize that a domain is defined by the consensus of its *stakeholders*, i.e. people having an interest in the domain, e.g. marketing and technical managers, programmers, end-users, and customers, and therefore it is subject to both politics and legacies.

Srinivas makes the key observation that the significance of a domain is *externally* attributed [Sri91]:

> *Nothing in the individual parts of a domain either indicates or determines the cohesion of the parts as a domain. The cohesion is external and arbitrary—a collection of entities is a domain only to an extent that it is* perceived *by a community as being useful for modeling some aspect of reality*.

Shapere explains this community-based notion of a domain as follows [Sha77] (paraphrase from [Sri91]):

> *In a given community, items of real-world information come to be associated as bodies of information or* problem domains *having the following characteristics:*
>
> - *deep or comprehensive relationships among the items of information are suspected or postulated with respect to some class of problems;*
> - *the problems are perceived to be significant by the members of the community*.

Finally, it is also important noting that the kinds of knowledge contained in a domain include both

- formal models, which can often be inconsistent among each other, e.g. different domain theories, and
- informal expertise, which is difficult or impossible to formalize (as exemplified by the problems in the area of expert systems [DD87]).

As a conclusion, we will adopt the following definition of a *domain*:

*Domain: An area of knowledge*

- *scoped to maximize the satisfaction of the requirements of its stakeholders,*
- *including a set of concepts and terminology understood by practitioners in that area, and*
- *including knowledge of how to build software systems (or parts of software systems) in that area.*

### 3.6.2 Domain Scope

*Horizontal and vertical scope*

There are two kinds of domain scope with respect to the software systems in a domain (see Figure 1, p. 7):

- *Horizontal scope or system category scope*: How many different systems are in the domain? For example, the domain of containers (e.g. sets, vectors, lists, maps, etc.) has a larger horizontal scope than the domain of matrices since more application need containers than matrices.
- *Vertical scope or per-system scope*: Which parts of these systems are in the domain? The vertical scope is the larger the larger parts of the systems are in the domain. For example, the vertical scope of the domain of containers is smaller than the vertical scope of the domain of portfolio management systems since containers capture only a small slice of the functionality of a portfolio management system.

*Vertical, horizontal, encapsulated, and diffused domains*

Based on the per-system scope, we distinguish between the following kinds of domains [SCK+96]:

- *vertical* vs. *horizontal* domains;
- *encapsulated* vs. *diffused* domains.

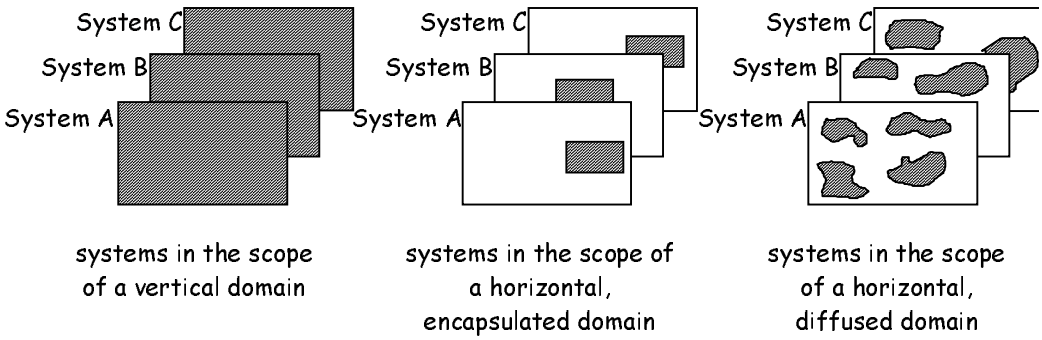



**Figure 9** *Vertical, horizontal, encapsulated, and diffused domains. (Each rectangle represents a system. The shaded areas depict system parts belonging to the domain.)*

Vertical domains contain complete systems (see Figure 9). Horizontal domains contain only parts of the systems in the domain scope. Encapsulated domains are horizontal domains where the system parts in the domain are well-localized with respect to their systems. Diffused domains are also horizontal domains, but they contain several, different parts of each system in the domain scope.[19]

*Native vs. innovative domains*

The scope of a domain can be determined using different strategies [SCK+96]:

1. choose a domain from the existing "*native*" domains (i.e. a domain which is already recognized in an organization);
2. define an *innovative* domain based on
   - a set of existing software systems sharing some commonalities (i.e. a family of systems) and/or
   - some marketing strategy.

*Product lines vs. product families*

The last two strategies are closely related to the following two concepts:

- *Product family*: "A product family is a group of products that can be built from a common set of assets." [Wit96, p. 16] A product family is defined on the basis of similarities between the structure of its member products. A product family shares at least

[19] Please note that, in Domain Engineering, the terms *horizontal* and *vertical* are used in a different sense than in the Object Management Architecture (OMA) defined by the Object Management Group (OMG, see www.omg.org). The OMG uses the term *vertical domain interfaces* to denote component interfaces specific to a specialized market (e.g. manufacturing, finance, telecom, transportation, etc.) and the term *horizontal facilities* (or *common facilities*) to denote generic facilities such as printing, database facilities, electronic mail facilities, etc. Thus, the OMG distinguishes between horizontal and vertical components, whereas in Domain Engineering we say that components have a horizontal nature in general since their scope does not cover whole systems but rather parts of systems. In Domain Engineering terms (see Section 6.4.1), OMG horizontal components are referred as *modeling components* (i.e. they model some general aspect such as persistency or printing) and the OMG vertical components are referred to as *application-specific components*. On the other hand, it is correct to say that modeling components have a larger horizontal scope than application-specific components.

a common generic architecture. *Product families are scoped based on commonalities between the products.*

- *Product line*: "A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market." [Wit96, p. 15] Thus, the definition of a product line is based on a marketing strategy rather than similarities between its member products. The features defined for a product line might require totally different solutions for different member products. A product line might be well served with one product family; however, it might also require more than one product family. On the other hand, a product family could be reused in more then one product line. *Product lines are scoped based on a marketing strategy.*

Unfortunately, the terms product family and product line are often used interchangeably in the literature.

We determine the scope of a domain during the domain scoping activity of Domain Analysis. The scope of a domain is influenced by several factors, such as the stability and the maturity of the candidate areas to become parts of the domain, available resources for performing Domain Engineering, and the potential for reuse of the Domain Engineering results within and outside an organization. In order to ensure a business success, we have to select a domain that strikes a healthy balance among these factors. An organization which does not have any experience with Domain Engineering should choose a small but important domain, e.g. some important aspect of most systems it builds. The resulting components and models can be reused on internal projects or sold outside the organization. After succeeding with the first domain, the organization should consider adding more and more domains to cover its product lines.

### 3.6.3 Relationships Between Domains

We recognize three major types of relationships between domains:

*Subdomains*

- A *is contained in* B: All knowledge in domain A also belongs to domain B, i.e. A is a *subdomain* of B.[20] For example, the domain of matrix packages is a subdomain of the domain of matrix computation packages since matrix computations cover both matrices and matrix computation algorithms.

*Support domains*

- A *uses* B: Knowledge in A references knowledge in B in a significant way, i.e. it is worthwhile to represent aspects of A in terms of B. We say that B is a *support domain* of A. For example, the storage aspect of a matrix package implemented using different containers from a container package. In other words, the domain of container packages is a support domain of the domain of matrix packages.

*Analogy domains*

- A *is analogous to* B [SCK+96]: There is a considerable amount of similarity between A and B; however, it is not necessarily worthwhile to express one domain in terms of the other. We say that A is an *analogy domain* of B. For example, the domain of numerical array packages is an analogy domain of the domain of matrix packages. They are both at a similar level of abstraction (in contrast to the more fundamental domain of containers, which could be a support domain for the domain of numerical array and the domain matrix packages) and clearly have different focuses (see Section 10.1.1.2.6). Yet still there is a considerable amount of similarity between them and studying one domain may provide useful insights into the other one.

[20] In [SCK+96] B is referred to as *generalization* of A and A as *specialization* of B.

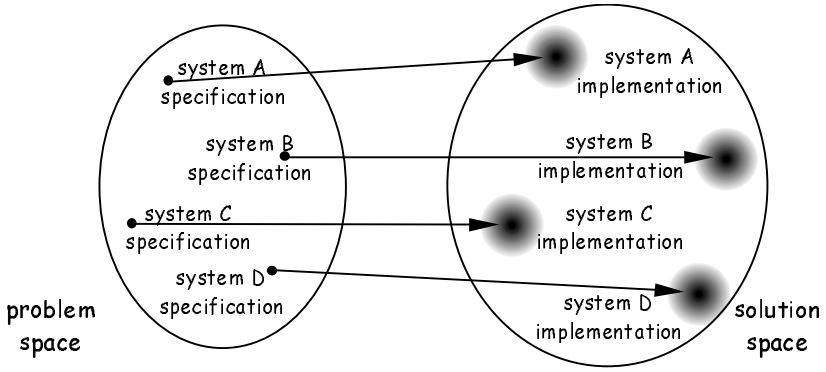



**Figure 10** *Problem and solution space*

### 3.6.4 Problem and Solution Space

The set of all valid system specifications in a domain (e.g. valid feature combinations) is referred to as the *problem space* and the set of all concrete systems in the domain is referred as to as the *solution space* (see Figure 10). One of the goals of Domain Engineering is to produce components, generators, production processes, etc., which automate the mapping between the system specifications and the concrete systems.

A problem space contains the domain concepts that application programmers wold like to interact with when specifying systems, whereas the solution space contains the implementation concepts. There is natural a tension between these two spaces because of their different goals: The domain concepts have a structure that allows direct and intentional expression of problems. On the other hand, when we design the implementation concepts, we strive for small, atomic components that can be combined in as many ways as possible. We want to avoid any code duplication by factoring out similar code sections into small, (parameterized) components. This is potentially at odds with the structure of the problem space since not all of these small components should be visible to the application programmer. There is a number of other issues to consider when we design both spaces. We discuss them in Section 9.4.3.

The overall structure of the solution space is referred to as the *target architecture*. For example, the target architecture of the generative matrix computation library described in Chapter 10 is a special form of a layered architecture referred to as the GenVoca architecture (see Section 6.4.2). The target architecture defines the framework for the integration of the implementation components.

*Domain-specific languages*

The system specifications in the problem space are usually expressed using a number of *domain-specific languages* (DSLs), i.e. languages specialized for the direct and declarative expression of system requirements in a given domain. These languages define the domain concepts. We discuss the advantages of DSLs in Section 7.6.1 and the issues concerning their design and implementation in Section 9.4.1.

### 3.6.5 Specialized Methods

Different kinds of systems require different modeling techniques. For example, most important aspects of interactive systems are captured by use cases and scenarios. On the other hand, large data-centric applications are sometimes more appropriately organized around entity-relationship diagrams or object diagrams. Additional, special properties such as real-time support, distribution, and high availability and fault tolerance require specialized modeling techniques. Thus, different categories of domains will require different specialized domain engineering methods, i.e. methods deploying specialized notations and processes. We will discuss this issue in Chapter 4. In Chapter 9, we present DEMRAL, a specialized Domain Engineering method for developing reusable algorithmic libraries.

## 3.7 Survey of Domain Analysis and Domain Engineering Methods

There is a large number of Domain Analysis and Domain Engineering methods. Two of them deserve special attention since they belong to the most mature and best documented (including case studies) methods currently available: *Feature-Oriented Domain Analysis* and *Organization Domain Modeling*. We describe them in Sections 3.7.1 and 3.7.2. Sections 3.7.3 through 3.7.8 contain short descriptions of twelve other Domain Engineering methods or approaches. Each of them has made important contributions to some aspects of Domain Engineering (such as conceptual clustering, rationale capture, formal approaches, etc.).

Two surveys of Domain Analysis methods have been published to date: [WP92] and the more comprehensive [Arr94]. Compared to these surveys, the following sections also reflect the newest development in the field of Domain Engineering.

Please note that, in order to be consistent with the original descriptions of the Domain Engineering methods in the literature, the survey uses the term *phase* in its older meaning, i.e. to denote *process components* (cf. footnote on page 33).

### 3.7.1 Feature-Oriented Domain Analysis (FODA)

*FODA* is a Domain Analysis method developed at the Software Engineering Institute (SEI). The method is described in [KCH+90]. Tool support for FODA is outlined in [Kru93] and a comprehensive example of applying FODA to the *Army Movement Control Domain* is described in [CSJ+92, PC91]. A number of other military projects to use FODA are listed in [CARDS94, p. F.2]. FODA has also been applied in the area of telecommunication systems, e.g. [Zal96, VAM+98].

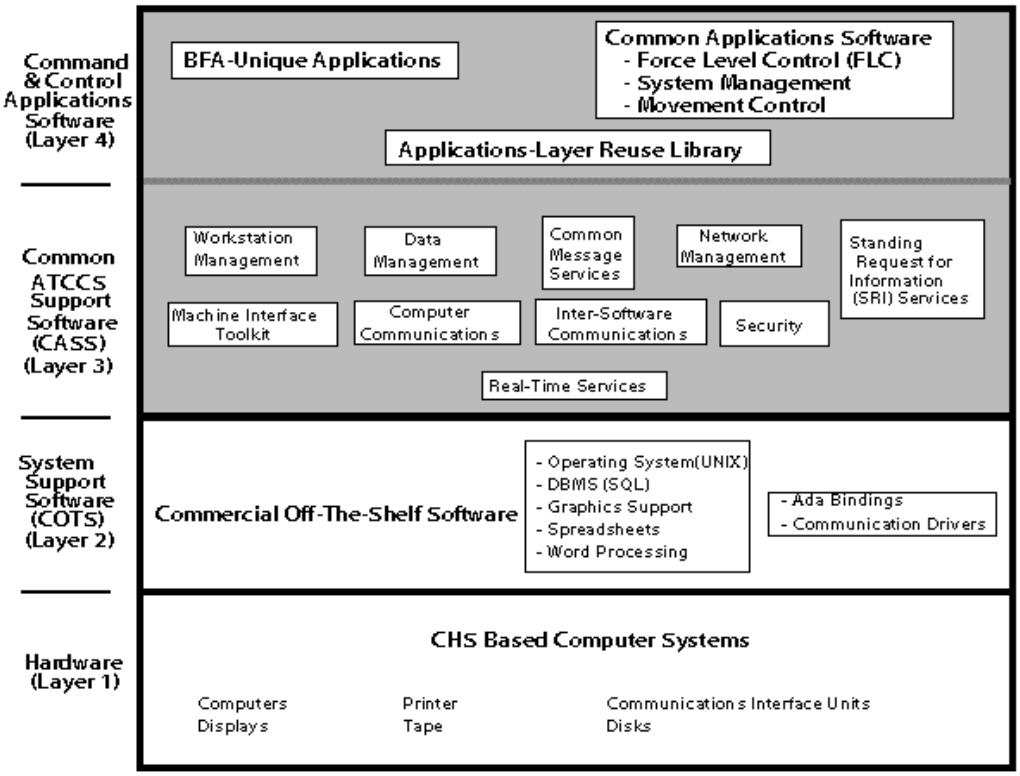



**Figure 11** *Example of a FODA structure diagram: The structure diagram of the Army Movement Control Domain (from [PC91])*

### 3.7.1.1 FODA Process

The FODA process consists of two phases [MBSE97]:[21]

1. *Context Analysis*: The purpose of Context Analysis is to define the boundaries of the domain to be analyzed.

2. *Domain Modeling*: The purpose of Domain Modeling is to produce a domain model.

We describe these phases in the following two subsections.

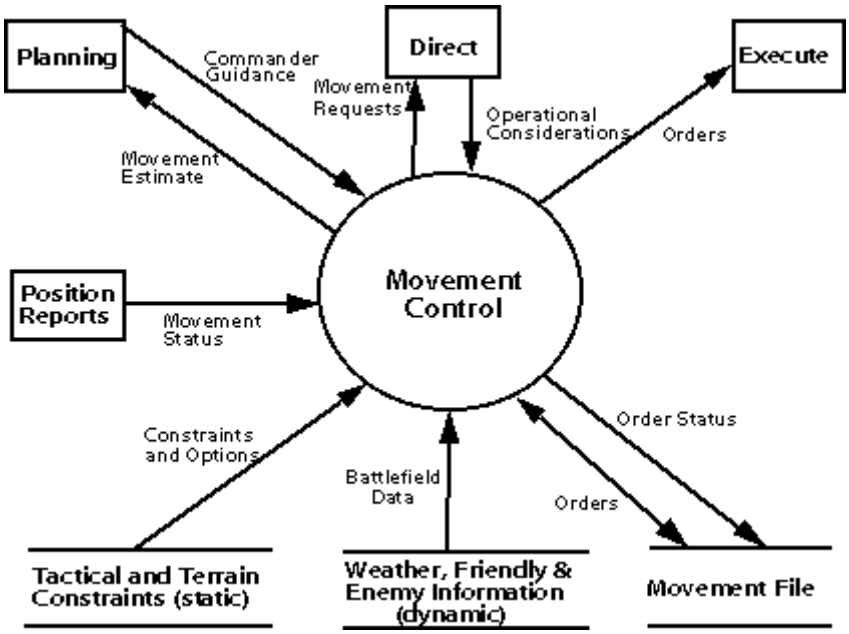



**Figure 12** *Example of a FODA context diagram: The context diagram of the Army Movement Control Domain (from [PC91]). A FODA context diagram is a typical data-flow diagram: "The arrows represent the information received or generated by the movement control domain. The closed boxes represent the set of sources and sinks of information. The open-ended boxes represent the databases that the movement control domain must interact with." [MBSE97]*

#### *3.7.1.1.1 Context Analysis*

The FODA Context Analysis defines the scope of a domain that is likely to yield useful domain products.[22] In this phase, the relationships between the domain of focus and other domains or entities are also established and analyzed for variability. The results of the context analysis along with factors such as availability of domain expertise and project constraints are used to limit the scope of the domain [MBSE97]. The results of the Context Analysis are the *context model* which includes a *structure diagram* (see Figure 11) and a *context diagram* (see Figure 12).

[21] Originally, FODA contained a third phase called *Architectural Modeling* (see [KCH+90]). This phase is no longer part of FODA, but instead it was converted into the *Domain Design* phase, which follows FODA in the overall framework of *Model-Based Software Engineering* (see Section 3.7.1.3).

[22] The FODA Context Analysis corresponds to the domain planning and domain scoping activities defined in Section 3.3.

#### *3.7.1.1.2 Domain Modeling*

During the FODA Domain Modeling phase the main commonalities and variabilities between the applications in the domain are identified and modeled. This phase involves the following steps [MBSE97]:

1. *Information Analysis*: The main purpose of Information Analysis is to capture domain knowledge in the form of domain entities and the relationships between them. The particular modeling technique used in this phase could be semantic networks, entity-relationship modeling, or object-oriented modeling. The result of Information Analysis is the *information model*, which corresponds to the concept model mentioned in Section 3.3.

2. *Features Analysis*: "Features Analysis captures a customer's or end-user's understanding of the general capabilities of applications in a domain. For a domain, the commonalities and differences among related systems of interest are designated as *features* and are depicted in the *features model*."[23] [MBSE97]

3. *Operational Analysis*: Operational Analysis yields the *operational model* which represents how the application works by capturing the relationships between the objects in the information model and the features in the features model.

Another important product of this phase is a *domain dictionary* which defines all the terminology used in the domain (including textual definitions of the features and entities in the domain).

### 3.7.1.2 The Concept of Features

In FODA, features are the properties of a system which *directly affect* end-users[24]:

*"Feature: A prominent and user-visible aspect, quality, or characteristic of a software system or systems."[KCH+90, p. 2]*

For example, "when a person buys an automobile a decision must be made about which transmission feature (e.g. *automatic* or *manual*) the car will have." [KCH+90, p. 35] Thus, FODA features can be viewed as features in the sense of Conceptual Modeling (see Section 2.2) with the additional requirement of directly affecting the end-user.

*Two definitions of feature*

In general there are two definitions of features found in Domain Engineering literature:

1. a end-user-visible characteristic of a system, i.e. the FODA definition, or

2. a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept. The latter definition is used in the context of ODM (see Section 3.7.2) and Capture (see Section 3.7.4) and is fully compatible with the understanding of features in Conceptual Modeling.

We prefer the latter definition since it is more general and covers the important case of software components.

*Mandatory, alternative, and optional features*

The features of a software system are documented in a *features model*. An important part of this model is the *features diagram*. An example of a simple features diagram of an automobile is shown in Figure 13. This example also illustrates three types of features[25]:

[23] The FODA term "features model" is equivalent to the term "feature model" defined in Section 3.3.

[24] A user may be a human user or another system with which applications in a domain typically interact.

1. *mandatory features*, which each application in the domain must have, e.g. all cars have a *transmission*;

2. *alternative features*, of which an application can posses only one at a time, e.g. *manual* or *automatic* transmission;

3. *optional features*, which an application may or may not have, e.g. *air conditioning*.

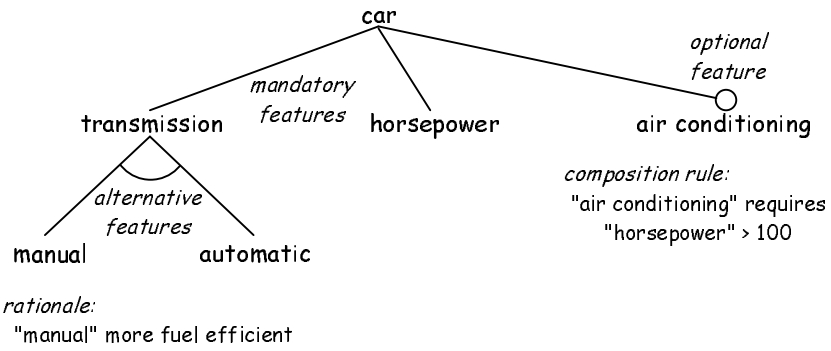
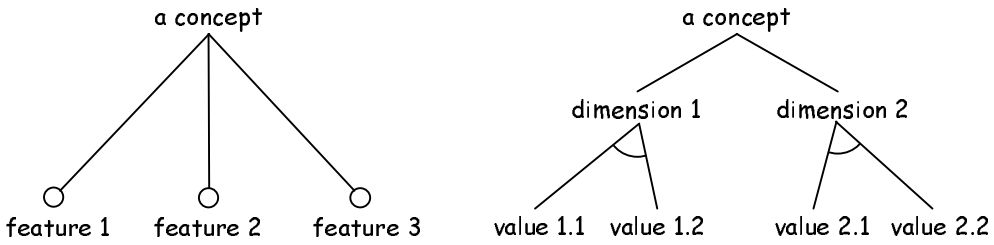



**Figure 13** *Example showing features of a car (from [KCH+90, p. 36]). Alternative features are indicated by an arc and optional features by an empty circle.*

The features diagram has the form of a tree in which the root represents the concept being described and the remaining nodes denote features. The relationships are *consists-of* relationships denoting, for example, that the description of a *transmission* consists of the descriptions of *manual* and *automatic* transmissions.

The FODA-style of featural description subsumes both the featural and the dimensional descriptions from the classical conceptual modeling, which we discussed in Sections 2.2.1 and 2.3.6. This is illustrated in Figure 14.




**a**. *featural description* **b**. *dimensional description*

**Figure 14** *Representation of featural and dimensional descriptions using FODA feature notation*

Feature interdependencies are captured using *composition rules* (see Figure 13). FODA utilizes two types of composition rules:

---

[25] Strictly speaking, we have to distinguish between direct features of a concept and subfeatues of features. Direct features of an application may be mandatory, alternative, or optional with respect to all applications within the domain. A subfeature may be mandatory, alternative, or optional with respect to only the applications which also have its parent feature. We explain this idea in Chapter 5.4.1.

*Composition rules*

1. *requires rules*: Requires rules capture implications between features, e.g. "*air conditioning* requires *horsepower* greater than 100" (see Figure 13).

2. *mutually-exclusive-with rules*: These rules model constraints on feature combinations. An example of such a rule is "*manual* mutually exclusive with *automatic*". However, this rule is not needed in our example since *manual* and *automatic* are alternative features. In general, mutually-exclusive-with rules allow us to exclude combinations of features where each feature may be seated in quite different locations in the feature hierarchy.

*Rationale*

We can also annotate features with *rationales*. A rationale documents the reasons or trade-offs for choosing or not choosing a particular feature. For example, manual transmission is more *fuel efficient* than automatic one. Rationales are necessary since, in practice, not all issues pertaining to the feature model can be represented formally as composition rules (due to the complexity involved or limited representation means). Theoretically, *fuel efficient* in Figure 13 could be modeled as a feature. In this case, the dependency between *manual* and *fuel efficient* could be represented as the following composition rule: *fuel efficient* requires *manual*. However, one quickly recognizes that the dependency between *fuel efficient* and *manual* is far more complex. First, we would need some measure of *fuel efficiency* and, second, *fuel efficiency* is influenced by many more factors than just the type of car transmission. The problem becomes similar to the problem of representing human expertise in expert systems [DD87]. Thus, stating the rationale informally allows us to avoid dealing with this complexity. In general, rationale refers to factors that are outside of the considered model.

*Two definitions of rationale*

The usage of the term rationale in the Domain Engineering literature is inconsistent. There are roughly two definitions of this term:

1. the trade-offs for choosing or not choosing a particular feature, i.e. the FODA definition (this notion is similar to the forces section in the description of a design pattern [GHJV95]);

2. the particular reason for choosing a specific feature after considering a number of trade-offs (this would correspond to recording the information about which forces were directly responsible for arriving at the decision made). The latter definition is used in Capture (Section 3.7.4) and in ODM (Section 3.7.2). This definition is motivated by the work on *design rationale capture* [Shu91, Bai97], the goal of which is to record the reason for selecting a particular design alternative by a (not necessarily software) designer during the design of a specific system.

Based on the purpose of a feature, the FODA features model distinguishes between *context*, *representation*, and *operational features* [MBSE97]:[26]

1. *Context features* "are those which describe the overall mission or usage patterns of an application. Context features would also represent such issues as performance requirements, accuracy, and time synchronization that would affect the operations." [MBSE97]

2. *Representation features* "are those features that describe how information is viewed by a user or produced for another application (i.e., what sort of input and output capabilities are available)." [MBSE97]

3. *Operational features* "are those features that describe the active functions carried out (i.e., what the application does)." [MBSE97]

Of course, other types of features are also possible. For example, Bailin proposes the following feature types: *operational*, *interface*, *functional*, *performance*, *development methodology*, *design*, and *implementation features* [Bai92].

---

[26] The original FODA description in [KCH+90] uses a slightly different categorization; it distinguishes between *functional*, *operational*, and *presentation* features.

Finally, FODA features are classified according to their binding time into *compile-time*, *activation-time*, and *runtime features* [KCH+90]:

1. *Compile-time features* are "features that result in different packaging of the software and, therefore, should be processed at compile time. Examples of this class of features are those that result in different applications (of the same family), or those that are not expected to change once decided. It is better to process this class of features at compile time for efficiency reasons (time and space)."

2. *Activation-time features* (or *load-time features*) are those "features that are selected at the beginning of execution but remain stable during the execution. [...] Software is generalized (e.g. table-driven software) for these features, and instantiation is done by providing values at the start of each execution."

3. *Runtime features* are those "features that can be changed interactively or automatically during execution. Menu-driven software is an example of implementing runtime features."

*Binding time, binding location, and binding site*

The FODA classification of features according to binding time is incomplete. There are also other times, e.g. linking time, or first-call time (e.g. when a method is called the first time; this time is relevant for just-in-time compilation [Kic97]). In general, feature *binding time* can be classified according to the specific times in the life cycle of a software system. Some specific products could have their specific times (e.g. debugging time, customization time, testing time, or, for example, the time when something relevant takes place during the use of the system, e.g. emergency time, etc.). Also, when a component is used in more than one location in a system, the allowed component features could depend on this *location*. Furthermore, binding could depend on the context or setting in which the system is used. For this reason, Simos et al. introduced the term *binding site* ([SCK+96]) which covers all these cases (i.e. binding time and context). We will discuss this concept in Section 5.4.4.3 in more detail.

The features model describes the problem space in a concise way: "The features model is the chief means of communication between the customers and the developers of new applications. The features are meaningful to the end-users and can assist the requirements analysts in the derivation of a system specification that will provide the desired capabilities. The features model provides them with a complete and consistent view of the domain." [MBSE97]

To summarize, a FODA features model consists of the following four key elements:

1. *features diagram*, i.e. a representation of a hierarchical decomposition of features including the indication whether or not each feature is mandatory, alternative, or optional;

2. *feature definitions* for all features including the indication of whether each feature is bound at compile time, activation time, or at runtime (or other times);

3. *composition rules* for features;

4. *rationale* for features indicating the trade-offs.

We will come back to this topic in Chapter 5, where we define a more comprehensive representation of feature models.

#### 3.7.1.3 FODA and Model-Based Software Engineering

FODA is a part of *Model-Based Software Engineering (MBSE)*, a comprehensive approach to family-oriented software engineering based on Domain Engineering, being developed by SEI

(see [MBSE97] and [Wit94]).[27] MBSE deploys a typical family-oriented process architecture consisting of two processes: Domain Engineering and Application Engineering (see Figure 8). The Domain Engineering process, in turn, consists of Domain Analysis, Domain Design, and Domain Implementation, where FODA takes the place of Domain Analysis.

### 3.7.2 Organization Domain Modeling (ODM)

*ODM* is a domain engineering method developed by Mark Simos of Synquiry Ltd. (formerly Organon Motives Inc.). The origins of ODM date back to Simos's work on the knowledge-based reuse support environment *Reuse Library Framework (RLF)* [Uni88]. Since then ODM has been used and refined on a number projects, most notably the STARS project (see Section 3.8), and other projects involving organizations such as Hewlett-Packard Company, Lockheed Martin (formerly Loral Defense Systems-East and Unisys Government Systems Group), Rolls-Royce, and Logicon [SCK+96]. During its evolution, ODM assimilated many ideas from other domain engineering approaches as well as work in non-software disciplines such as organization redesign and workplace ethnography [SCK+96]. The current version 2.0 of ODM is described in [SCK+96], a comprehensive guidebook comprising almost five hundred pages. This guidebook replaces the original ODM description in [SC93].

*Unique aspects of ODM*

Some of the unique aspects of ODM include

- *Focus on stakeholders and settings*: Any domain concepts and features defined during ODM have explicit traceability links to their stakeholders and relevant contexts (i.e. settings). In addition, ODM introduces the notion of a *grounded* abstraction, i.e. abstraction based on stakeholder analysis and setting analysis, as opposed to the "right" abstraction (a term used in numerous textbooks on software design), which is based on intuition.

- *Types of domains*: ODM distinguishes between horizontal vs. vertical, encapsulated vs. diffused, and native vs. innovative domains (see Sections 3.6.1 and 3.6.2).

- *More general notion of feature*: ODM uses a more general notion of feature than FODA (see Section 3.7.1.2). An ODM feature does not have to be end-user visible; instead, it is defined as a difference between two concepts (or variants of a concept) that "makes a significant difference" to some stakeholder. ODM features directly correspond to the notion of features discussed in Chapter 2.

- *Binding site*: In FODA, a feature can be bound at compile, start, or runtime (see Section 3.7.1.2). ODM goes beyond that and introduces the notion of *binding site*, which allows for a broader and finer classification of binding times and contexts depending on domain-specific needs. We discuss this idea in Section 5.4.4.3

- *Analysis of feature combinations*: ODM includes explicit activities aimed towards improving the quality of features, such as feature clustering (i.e. co-occurrence of features), as well as the building of a closure of feature combinations (i.e. enumerating all valid feature combinations). The latter can lead to the discovery of innovative system configurations which have not been considered before.

- *Conceptual modeling*: ODM uses a very general modeling terminology similar to that introduced in Chapter 2. Therefore, ODM can be specialized for use with any specific system modeling techniques and notations, such as object-oriented analysis and design (OOA/D) methods and notations or structured methods. We discuss this topic in Chapter 4. Also, in Chapter 9, we present a specialization of ODM for developing algorithmic libraries.

- *Concept starter sets*: ODM does not prescribe any particular concept categories to look for during modeling. While other methods specifically concentrate on some concept

[27] FODA was conceived before the work on MBSE started.

categories such as objects, functions, algorithms, data structures, etc., ODM uses *concept starter sets* consisting of different combinations of concept categories to jumpstart modeling in different domains.

- *Scoping of the asset base*: ODM does not require the implementation of the full domain model. There is an explicit ODM task, the goal of which is to determine the part of the domain model to be implemented based on project and stakeholder priorities.

- *Flexible architecture*: ODM postulates the need for a *flexible architecture* since a *generic architecture* is not sufficient for domains with a very high degree of variability (see Section 3.4).

- *Tailorable process*: ODM does not commit itself to any particular system modeling and engineering method, or any market analysis, or any stakeholder analysis method. For the same reason, the user of ODM has to provide these methods, select appropriate notations and tools (e.g. feature notation, object-oriented modeling, etc.), and also invest the effort of integrating them into ODM.

The following section gives a brief overview of the ODM process.

#### 3.7.2.1 The ODM Process

The ODM process—as described in [SCK+96]—is an extremely elaborate and detailed process. It consists of three main phases:

1. *Plan Domain*: This is the domain scoping and planning phase (Section 3.3) corresponding to Context Analysis in FODA (Section 3.7.1.1.1).

2. *Model Domain*: In this phase the *domain model* is produced. It corresponds to Domain Modeling in FODA (3.7.1.1.2).

3. *Engineer Asset Base*: The main activities of this phase are to produce the architecture for the systems in the domain and to implement the reusable assets.

Plan Domain and Model Domain clearly correspond to a typical Domain Analysis. Engineer Asset Base corresponds to Domain Design and Domain Implementation.

Each of the three ODM phases consists of three sub-phases and each sub-phase is further divided into three tasks. The complete ODM process is shown in Figure 15.

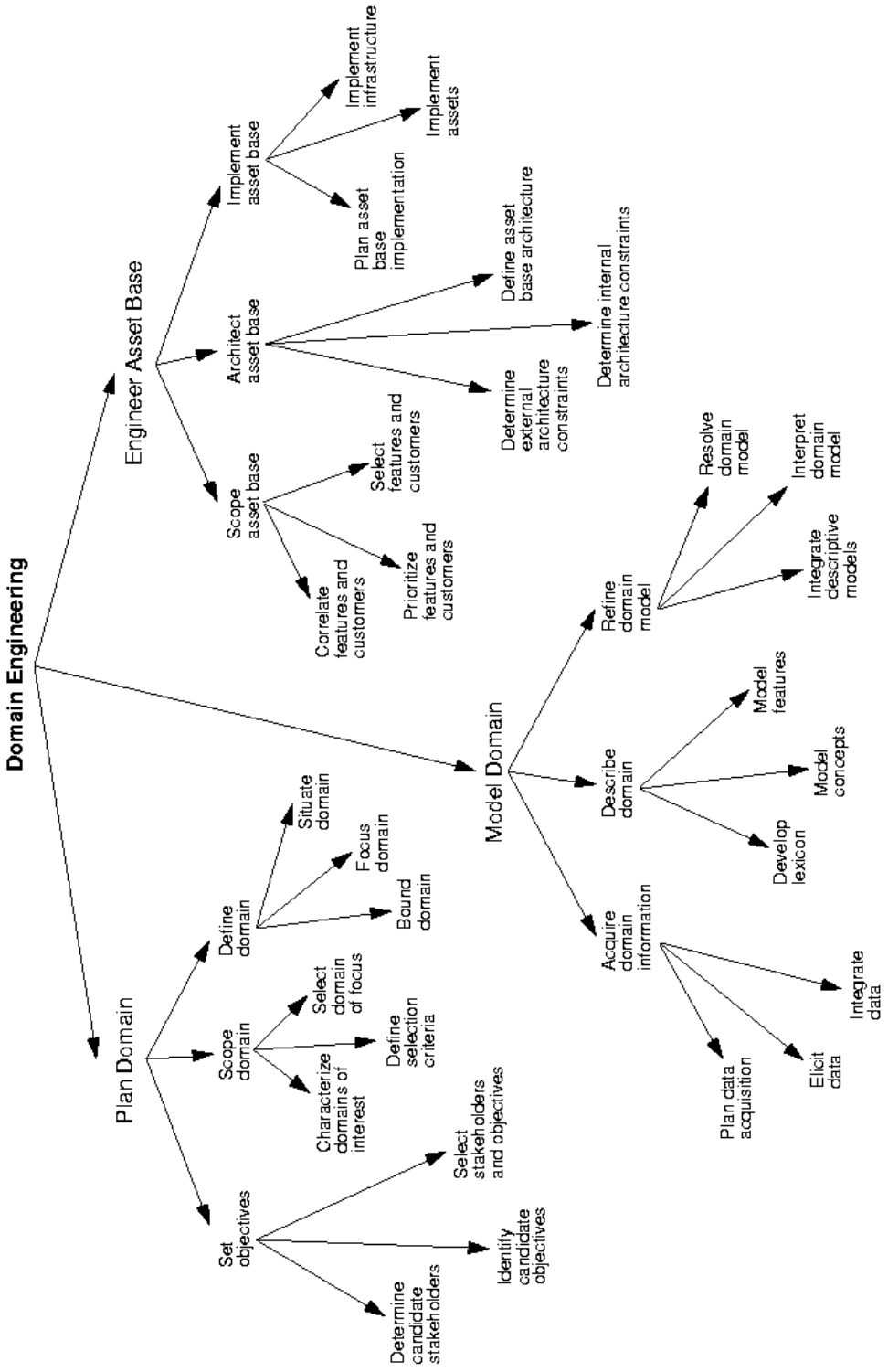



**Figure 15** *Phases of the ODM process (from [SCK+96, p. 40])*

The ODM phases and sub-phases are described in Table 5.

| ODM Phase | ODM Sub-Phase | Performed Tasks[28] |
|---|---|---|
| Plan Domain | Set objectives | • determine the stakeholders (i.e. any parties related to the project), e.g. *end-users, customers, managers, third-party suppliers, domain experts, programmers, subcontractors*<br>• analyze stakeholders' objectives and project objectives<br>• select stakeholders and objectives from the candidates |
| | Scope domain | • scope the domain based on the objectives (issues include choosing between *vertical* vs. *horizontal*, *encapsulated* vs. *diffused*, *native* vs. *innovative* domains) |
| | Define domain | • define the domain boundary by giving examples of systems in the domain, counterexamples (i.e. systems outside the domain), as well as generic rules defining what is in the domain and what not<br>• identify the main features of systems in the domain and the usage settings (e.g. development, maintenance, customization contexts) for the systems<br>• analyze the relationships between the domain of focus and other domains |
| Model Domain | Acquire domain information | • plan the domain information acquisition task<br>• collect domain information from domain experts, by reverse-engineering existing systems, literature studies, prototyping, etc.<br>• integrate the collected data, e.g. by pre-sorting the key domain terms, identifying the most important system features |
| | Describe domain | • develop a lexicon of domain terms<br>• model the semantics of the key domain concepts<br>• model the variability of concepts by identifying and representing their features |
| | Refine domain | • integrate the models produced so far into an overall consistent model<br>• model the rationale for variability, i.e. the trade-offs for using or not using certain features<br>• improve the quality of features by clustering and experimenting with innovative feature combinations |
| Engineer Asset Base | Scope asset base | • correlate identified features and customers<br>• prioritize features and customers<br>• based on the priorities, select the portion of the modeled functionality for implementation |
| | Architect asset base | • determine external architecture constraints (e.g. external interfaces and the allocation of features to the external interfaces)<br>• determine internal architecture constraints (e.g. internal interfaces, allocation of groups of related features to internal interfaces)<br>• define asset base architecture based on these constraints |
| | Implement asset base | • plan asset base implementation (e.g. selection of tools, languages, and other implementation strategies)<br>• implement assets<br>• implement infrastructure (e.g. domain-specific extensions to general infrastructures, asset retrieval mechanisms, asset qualification mechanisms) |

**Table 5** *Description of ODM phases and sub-phases*

[28] The tasks listed in this column do not exactly correspond to the formal ODM tasks. The latter are shown in Figure 15.

### 3.7.3 Draco

*Draco* is an approach to Domain Engineering as well as an environment based on transformation technology. Draco was developed by James Neighbors in his Ph.D. work [Nei80] to be the first Domain Engineering approach. Furthermore, the main ideas introduced by Draco include *domain-specific languages* and *components as sets of transformations*. This section gives a brief overview of Draco. A more detailed discussion is given in Section 6.4.1.

The main idea of Draco is to organize software construction knowledge into a number of related domains. Each Draco domain encapsulates the needs and requirements and different implementations of a collection of similar systems. Specifically, a Draco domain contains the following elements ([Nei84, Nei89]):

- *Formal domain language (also referred to as "surface" language)* : The domain language is used to describe certain aspects of a system. The domain language is implemented by a parser and a pretty printer. The internal form of parsed code is a *parse tree*. The term *domain language* is equivalent to the term *domain-specific language* introduced in Section 3.6.4.

- *Set of optimization transformations*: These transformations represent rules of exchange of equivalent program fragments in the domain language and are useful for performing optimizations on the parse tree.

- *Set of transformational components*: Each component consists of one or more *refinement transformations* capable of translating the objects and operations of the source domain language into one or more target domain languages of other, underlying domains. There is one component for each object and operation in the domain. Thus, components implement a program in the source domain language in terms of the target domains. Draco refers to the underlying target domains as *refinements* of the source domain. As a result, the construction knowledge in Draco is organized into domains connected by *refinement* relationships.

- *Domain-specific procedures*: Domain-specific procedures are used whenever a set of transformations can be performed algorithmically. They are usually applied to perform tasks such as generating new code in the source domain language or analyzing programs in the source language.

- *Transformation tactics and strategies (also called optimization application scripts)*: Tactics are domain-independent and strategies are domain-dependent rules helping to determine when to apply which refinement. Optimizations, refinements, procedures, tactics, and strategies are organized into metaprograms (i.e. programs generating other programs).

It is important to note that, in Draco, a system is represented by many domain languages simultaneously.

The results of applying Draco to the domain of real-time applications and the domain of processing standardized tests are described in [Sun83] and [Gon81], respectively.

### 3.7.4 Capture

*Capture*, formerly known as KAPTUR (see [Bai92, Bai93]), is an approach and a commercial tool for capturing, organizing, maintaining, and representing domain knowledge. Capture was developed by Sidney Bailin of CTA Inc. (currently with Knowledge Evolution Inc.).

The Capture tool is a hypertext-based tool allowing the user to navigate among assets (e.g. architectures and components). The assets are documented using informal text and various diagrams, such as entity-relationship diagrams. The assets are annotated by their *distinctive features*, which document important design and implementation decisions. Features are

themselves annotated with *trade-offs* that were considered and *rationale* for the particular decision made.[29] [Bai92]

### 3.7.5 Domain Analysis and Reuse Environment (DARE)

*DARE* is both a Domain Analysis method and a tool suite supporting the method [FPF96]. DARE was developed by William Frakes (Software Engineering Guild) and Rubén Prieto-Díaz (Reuse Inc.) and represents a commercial product.

*Clusters and facets*

The DARE tool suite includes lexical analysis tools for extracting domain vocabulary from system descriptions, program code, and other sources of domain knowledge. One of the most important tools is the *conceptual clustering* tool, which clusters words according to their conceptual similarity. The clusters are further manually refined into *facets*, which are main categories of words and phrases that fall in the domain [FPF96]. The idea of using facets to describe and organize systems and components in a domain has its roots in the application of library science techniques, such as faceted classification, to component retrieval [Pri85, Pri87, PF87, Pri91a, Pri91b, OPB92].

The main workproducts of DARE include a *facet table*, *feature table*, *system architecture*, and *domain lexicon* and are organized into a *domain book*. The DARE tool suite includes appropriate tools for creating and viewing these workproducts.

### 3.7.6 Domain-Specific Software Architecture (DSSA) Approach

The *DSSA* approach to Domain Engineering was developed under the *Advanced Research Project Agency's (ARPA) DSSA Program* (see [Hay94, TTC95]). The DSSA approach emphasizes the central role of the concept of *software architecture* in Domain Engineering. The overall structure of the DSSA process is compatible with the generic process structure described in Sections 3.3 through 3.5 (see [CT93, TC92] for descriptions of the DSSA process). The main workproducts of the DSSA process include the following [Tra95]:

1. *Domain Model*: The DSSA Domain Model corresponds to the concept model in Section 3.3 (i.e. concept model in ODM or information model in FODA) rather than a full domain model.

2. *Reference Requirements*: The DSSA Reference Requirements are equivalent to the feature model in Section 3.3. Each *reference requirement* (or *feature* in the terminology of Section 3.3) is either mandatory, optional, or alternative. The DSSA Reference Requirements include both functional and non-functional requirements.[30]

3. *Reference Architecture*: A DSSA Reference Architecture is an architecture for a family of systems consisting mainly of an *architecture model*, *configuration decision tree* (which is similar to the FODA features diagram in Section 3.7.1.2), *design record* (i.e. description of the components), and *constraints* and *rationale* (the latter two correspond to configuration rules and rationale in FODA in Section 3.7.1.2).

The need to formally represent the components of an architecture and their interrelationships led to the development of so-called *Architecture Description Languages* or *ADLs*. The concept of ADLs is described in [Cle96, Arch, SG96].

The DSSA approach has been applied to the avionics domain under the *Avionics Domain Application Generation Environment (ADAGE)* project involving Loral Federal Systems and other contractors (see [ADAGE]). As a result of this effort, a set of tools and other products supporting the DSSA process have been developed, including the following [HT94]:

---

[29] Note that this terminology is different from the FODA terminology, according to which *rationale* and *trade-offs* are synonyms (see Section 3.7.1.2).

[30] Note that the DSSA Reference Requirements are not part of the DSSA Domain Model, whereas the feature model is part of the domain model in Section 3.3.

- *DOMAIN*: a hypermedia-based Domain Analysis and requirements capture environment;
- *MEGEN*: an application generator based on module expressions;
- *LILEANA*: an ADL based on the ADA annotation language ANNA [LHK87] and the module interconnection language LIL [Gog83] (LILEANA is described in [Tra93, GT96]).

Other DSSA program efforts resulted in the development of other Domain Engineering tools and products (see [HT94] for more details), most notably the ADLs *ArTek* (developed by Teknowledge [THE+94]), *ControlH* and *MetaH* (developed by Honeywell [BEJV93]), and *Rapide* (developed at Stanford University [LKA+95]).

### 3.7.7 Algebraic Approach

The *algebraic approach* to Domain Engineering was proposed by Yellamraju Srinivas in [Sri91] (see [Smi96, SJ95] for more recent work). This section gives a brief overview of this approach. A more detailed description follows in Section 6.4.4.

The main idea of this approach is to formalize domain knowledge in the form of a network of related *algebraic specifications* (also referred to as *theories*). An algebraic specification defines a *language* and constrains its possible meanings through *axioms* and *inference rules*. Algebraic specifications can be related using *specification morphisms*. Specification morphisms define translations between specification languages that preserve the *theorems* (i.e. all statements which can be derived from the axioms using the inference rules). Thus, in the algebraic approach, the domain model is represented as a number of formal languages including translations between them. From this description, it is apparent that the algebraic approach and the Draco approach (Section 3.7.3) are closely related.[31] In fact, the only difference is that the algebraic approach is based on the algebraic specification theory (e.g. [LEW96]) and the category theory (e.g. [BW85]). Similarly to Draco, the algebraic approach lends itself well to implementation based on transformations. The inference rules of a specification correspond to the optimization transformations of Draco, and the specification morphisms correspond to refinement transformations.

First success reports on the practical use of the algebraic approach include the application of the transformation-based system *KIDS* (*Kestrel Interactive Development System*, see [Smi90]) in the domain of *transportation scheduling* by the Kestrel Institute. According to [SPW95], the scheduler generated from a formal domain model using KIDS is over 20 times faster than the standard, hand-coded system deployed by the customer. This proves the viability of the algebraic approach in narrow, well-defined domains. A successor system to KIDS is SPECWARE [SJ95], which is explicitly based on category theory (i.e. it uses category theory concepts both in its design and user interface).

### 3.7.8 Other Approaches

Other approaches to Domain Engineering include the following:

- *SYNTHESIS*: SYNTHESIS [SPC93] is a Domain Engineering method developed by the Software Productivity Consortium in the early nineties. The structure of the SYNTHESIS process is principally consistent with the generic process structure described in Sections 3.3 through 3.5 (although it uses a slightly different terminology). A unique aspect of SYNTHESIS is the tailorability of its process according to the levels of the *Reuse Capability Model* [SPC92]. This tailorability allows an organization to control the impact of the reuse process installation on its own structures and processes.

[31] As indicated in [Sri91, p. 91], the work on Draco has had a major influence on the algebraic approach to Domain Engineering.

- *Family-Oriented Abstraction, Specification, and Translation (FASST)*: FASST is a Domain Engineering method developed by David Weiss et al. at Lucent Technologies Bell Laboratories [Wei96]. FASST has been greatly influenced by the work on SYNTHESIS (Weiss was one of the developers of SYNTHESIS).

- *Defense Information Systems Agency's Domain Analysis and Design Process (DISA DA/DP)*: DISA DA/DP [DISA93] is similar to MBSE (Section 3.7.1.3) and ODM (Section 3.7.2). However, it only includes Domain Analysis and Domain Design. DISA DA/DP uses the object-oriented Coad-Yourdon notation [CY90].

- *Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA)*: JODA [Hol93] is a Domain Analysis method similar to FODA (see Section 3.7.1; however, JODA does not include a feature model) and is based on the object-oriented Coad-Yourdon notation and analysis method [CY90].

- *Gomaa*: [Gom92] describes an early object-oriented Domain Engineering method developed by Hassan Gomaa. An environment supporting the method is set out in [GKS+94].

- *Reusable Ada Products for Information Systems Development (RAPID)*: RAPID is a Domain Analysis approach developed by Vitaletti and Guerrieri [VG90], utilizing a similar process to the afore-presented Domain Engineering methods.

- *Intelligent Design Aid (IDeA)*: IDeA is a design environment supporting Domain Analysis and Domain Design [Lub91]. IDeA was developed by Mitchell Lubars. The unique aspect of IDeA is its iterative approach to Domain Analysis, whereby specific problems are analyzed one at a time and each analysis potentially leads to an update of the domain model.[32]

Since the main concepts and ideas of Domain Engineering have already been illustrated based on the methods presented in previous sections, we refrain from describing the approaches mentioned in this section in more detail.

## 3.8 Historical Notes

The idea of Domain Engineering can be traced back to the work on program families by Dijkstra [Dij70] and Parnas [Par76]. Parnas defines *program family* as follows [Par76, p. 1]:

> *"We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members."*

The term *Domain Analysis* was first defined by Neighbors in his Ph.D. work on Draco [Nei80, pp. xv-xvi] as

> *"the activity of identifying objects and operations of a class of similar systems in a particular problem domain."*

Major efforts aimed at developing Domain Analysis methods (including SEI's FODA and the work by Prieto-Diaz et al. at the Software Productivity Consortium) followed in the late eighties. A comprehensive bibliography of work related to Domain Engineering from the period 1983-1990 can be found in [HNC+90].

A large share of the work on Domain Engineering was sponsored by the U.S. Department of Defense research programs related to software reuse including *Software Technology for*

---

[32] In [Lub91] the term *domain engineering* is defined as the phase in which reusable assets identified during *domain analysis* are constructed. This terminology is inconsistent with the terminology currently recognized in the Domain Engineering community.

*Adaptable, Reliable Systems (STARS)* [STARS94], *Comprehensive Approach to Reusable Defense Software (CARDS)* [CARDS], and DSSA (Section 3.7.6).

Domain Engineering methods such as MBSE, ODM 2.0, and FASST (Sections 3.7.1.3, 3.7.2, 3.7.8) can be classified as second generation methods. The most recent trend in the field is to integrate Domain Engineering and OOA/D methods (see Chapter 4).

A partial genealogy of Domain Engineering methods is shown in Figure 16.

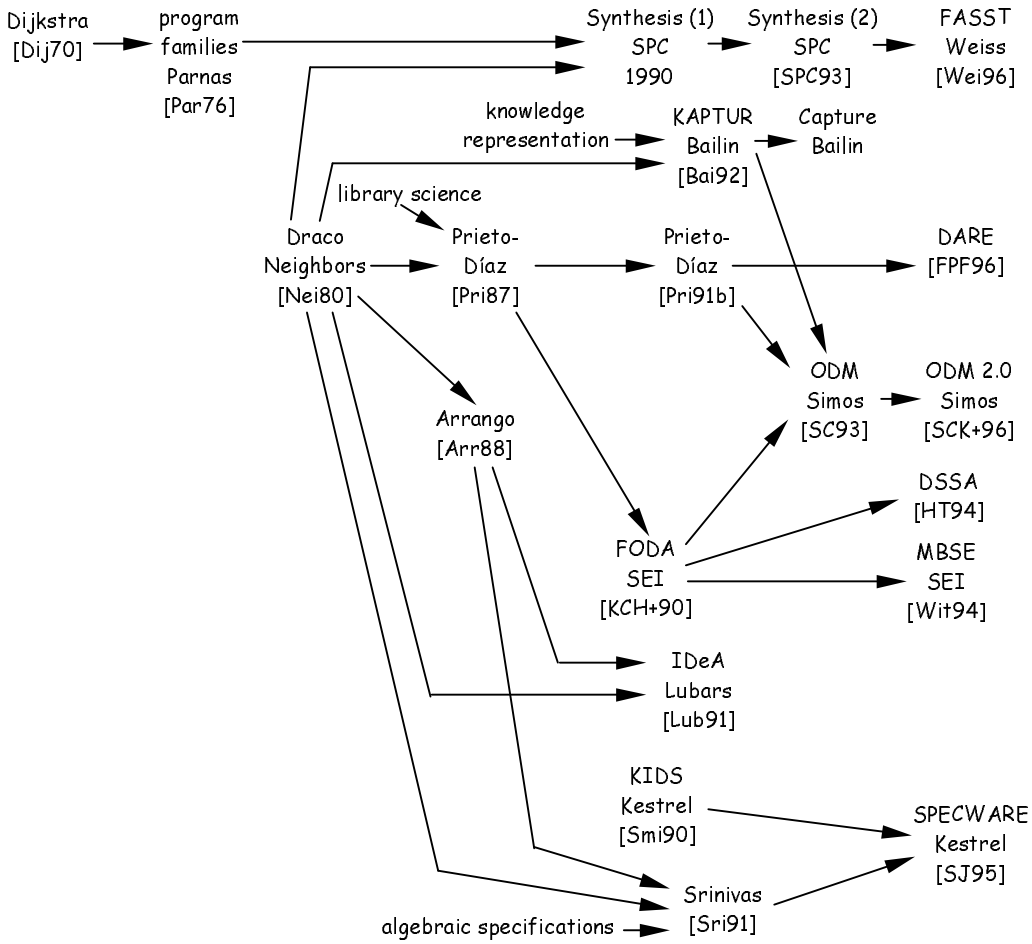



**Figure 16** *Partial genealogy of Domain Engineering (based on [FP96])*

## 3.9 Discussion

Domain Engineering represents a valuable approach to software reuse and multi-system-scope engineering. Table 6 compares conventional software engineering and Domain Engineering based on their workproducts. Another important difference is the split of software engineering into *engineering for reuse* (Domain Engineering) and *engineering with reuse* (Application Engineering).

Domain Engineering moves the focus from code reuse to reuse of analysis and design models. It also provides us with a useful terminology for talking about reuse-based software engineering.

| Software Engineering | Domain Engineering |
|---|---|
| *Requirements Analysis*<br>↷ requirements for one system | *Domain Analysis*<br>↷ reusable requirements for a class of systems |
| *System Design*<br>↷ design of one system | *Domain Design*<br>↷ reusable design for a class of systems |
| *System Implementation*<br>↷ implemented system | *Domain Implementation*<br>↷ reusable components, infrastructure, and production process |

**Table 6** *Comparison between conventional software engineering and Domain Engineering*

Based on the discussion of the various Domain Engineering methods, we arrive at the following conclusions:

1. The described methods and approaches are quite similar regarding the process. They use slightly different terminology and different groupings of activities, but they are, to a large degree, compatible with the generic process described in Sections 3.3 through 3.5. This process is also well exemplified by MBSE and ODM (Sections 3.7.1 and 3.7.2).

2. However, as the overall process framework remains quite stable, significant variations regarding the concrete modeling techniques and notations, approaches to software architecture, and component implementation techniques are possible. In particular, questions regarding the relationship between Domain Engineering and object-oriented technology are interesting. How do OOA/D and Domain Engineering fit together? We will address this topic in Chapter 4. Furthermore, we need to look for adequate technologies for implementing domain models. We will discuss some implementation technologies in Chapter 6 and Chapter 7.

3. Some of the presented Domain Engineering approaches make specific contributions with respect to the issue of concrete modeling techniques and implementation technologies:

   - As exemplified by Draco and the algebraic approach (Sections 3.7.3 and 3.7.7), formal methods, formal domain-specific languages, and transformation systems are well suited for mature and narrow domains. More work is needed in order to investigate a broader scope of applicability of these techniques.

   - The importance of informal techniques has also been recognized, e.g. the application of hypermedia systems for recording requirements, rationale, and informal expertise (Capture, Section 3.7.4), and the utilization of lexical analysis for domain vocabulary extraction (DARE, Section 3.7.5).

4. Which modeling techniques are most appropriate depends on the kind of the domain. For example, important aspects of GUI-based applications are captured by use cases and scenarios, whereas in scientific computing, algorithms are best captured using pseudocode. Furthermore, if the applications have some special properties, such as real-time aspects or distribution aspects, we need to apply additional, specialized modeling techniques. The organization of the Domain Engineering process itself depends on the organization and its business objectives. Thus, there will not be one Domain Engineering method appropriate for all possible domains and organizations. We will rather have specialized methods for different kinds of domains with tailorable processes for different organizational needs.

5. A major problem of all the existing Domain Engineering methods is that they do not address the evolution aspect of Domain Engineering. In [Arr89], Arrango emphasized that Domain Engineering is a continuous learning process, in which each new experience

in building new applications based on the reusable models produced in the Domain Engineering process is fed back into the process, resulting in the adjustment of the reusable models. None of the existing methods properly addresses these issues. They rather address only one full Domain-Engineering cycle and do not explain how to organize an efficient iterative process. The aspect of learning is usually treated as part of the reuse infrastructure, i.e. the results of using an asset should be fed back into the asset base. But since the reuse infrastructure is a product of Domain Engineering, its feed-back aspect is detached from the Domain Engineering process itself.

6. The methods also do not address how and when the infrastructure and the application production process are planned, designed, and implemented (only to include an infrastructure implementation activity in Domain Implementation is clearly insufficient).

## 3.10 References

[ADAGE] WWW home page of the DSSA/ADAGE project at http://www.owego.com/dssa/

[Arch] WWW home page of the Architecture Resources Guide at http://www-ast.tds-gn.lmco.com/arch/guide.html

[Arr88] G. Arango. Domain Engineering for Software Reuse. Ph.D. Dissertation, Department Information and Computer Science, University of California, Irvine, California, 1988

[Arr89] G. Arrango. Domain Analysis: From Art Form to Engineering Discipline. In *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 3, May 1989, pp. 152-159

[Arr94] G. Arrango. Domain Analysis Methods. In *Software Reusability*, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, 1994, pp. 17-49

[Bai92] S. Bailin. KAPTUR: A Tool for the Preservation and Use of Engineering Legacy. CTA Inc., Rockville, Maryland, 1992, http://www-ast.tds-gn.lmco.com/arch/kaptur.html

[Bai93] S. Bailin. Domain Analysis with KAPTUR. In *Tutorials of TRI-Ada'93*, Vol. I, ACM, New York, New York, September 1993

[Bai97] S. Bailin. Applying Multi-Media to the Reuse of Design Knowledge. In the *Proceedings of the Eighth Annual Workshop on Software Reuse*, 1997, http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers.html

[BC96] L. Brownsword and P. Clements. A Case Study in Successful Product Line Development. Technical Report, SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, October, 1996 , http://www.sei.cmu.edu

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998

[BCR94] V. Basili, G. Caldiera and D. Rombach, The Experience Factory. In *Encyclopedia of Software Engineering*, Wiley, 1994, ftp://ftp.cs.umd.edu/pub/sel/papers/fact.ps.Z

[BEJV93] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control, Honeywell Technology Center, 1993, http://www-ast.tds-gn.lmco.com/arch/dssa.html

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, Chichester, UK, 1996

[BP89] T. Biggerstaff and A. Perlis. *Software Reusability. Volume I: Concepts and Models*. ACM Press, Frontier Series, Addison-Wesley, Reading, 1989

[Buc97] S. Buckingham Shum. Negotiating the Construction and Reconstruction of Organisational Memories. In *Journal of Universal Computer Science*, Special Issue on IT for Knowledge Management, vol. 3, no. 8, 1997, pp. 899-928, http://www.iicm.edu/jucs_3_8/negotiating_the_construction_and/, also see http://kmi.open.ac.uk/~simonb/DR.html

[BW85] M. Barr and C. Wells. Toposes, triples and theories. Grundlagen der mathematischen Wissenschaften, vol. 278, Springer-Verlag, New York, New York, 1985

[CARDS] WWW home page of the Comprehensive Approach to Reusable Defense Software (CARDS) Program at http://www.cards.com

[CARDS94] Software Technology For Adaptable, Reliable Systems (STARS). Domain Engineering Methods and Tools Handbook: Volume I — Methods: Comprehensive Approach to Reusable Defense Software (CARDS). STARS Informal Technical Report, STARS-VC-K017R1/001/00, December 31, 1994, http://nsdir.cards.com/libraries/HTML/CARDS_documents.html

[Cle96] P. Clements. A Survey of Architecture Description Languages. In *Proceedings of Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996

[CK95] P. Clements and P. Kogut. Features of Architecture Description Languages. In *Proceedings of the 7th Annual Software Technology Conference*, Salt Lake City UT, April 1995

[Con97] E. Conklin. Designing Organizational Memory: Preserving Intellectual Assets in a Knowledge Economy. Technical Note, Group Decision Support Systems, http://www.gdss.com/DOM.htm, 1997

[CSJ+92] S. Cohen, J. Stanley, S. Peterson, and R. Krut. Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain. Technical Report, CMU/SEI-91-TR-28, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992, http://www.sei.cmu.edu

[CT93] L. Coglianese and W. Tracz. Architecture-Based Development Guidelines for Avionics Software. Version 2.1, Technical Report, ADAGE-IBM-92-03, 1993

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990

[DD87] H. Dreyfus and S. Dreyfus. *Künstliche Intelligenz: von den Grenzen der Denkmaschine und dem Wert der Intuition*. Rowohlt, rororo Computer 8144, Reinbek, 1987

[Dict] *The American Heritage Dictionary of the English Language*. Third Edition, Houghton Mifflin Company, 1992

[Dij70] E. Dijkstra. Structured Programming. In *Software Engineering Techniques*, J. Buxton and B. Randell, (Eds.), NATO Scientific Affairs Division, Brussels, Belgium, 1979, pp. 84-87

[DISA93] DISA/CIM Software Reuse Program. Domain Analysis and Design Process, Version 1. Technical Report 1222-04-210/30.1, DISA Center for Information Management, Arlington Virginia, March 1993

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998)*. IEEE Computer Society Press, 1998

[EP98] H.-E. Eriksson and M. Penker. *UML Toolkit*. John Wiley & Sons, 1998

[FP96] B. Frakes and R. Prieto-Díaz. Introduction to Domain Analysis and Domain Engineering. Tutorial Notes, The Fourth International Conference on Software Reuse, Orlando, Florida, April 23-26, 1996

[FPF96] W. Frakes, R. Prieto-Díaz, and Christopher Fox. DARE: Domain Analysis and Reuse Environment. Draft submitted for publication, April 7, 1996

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GKS+94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli. A Prototype Domain Modeling Environment for Reusable Software Architectures. In *Proceedings of the Third International Conference on Software Reuse*, Rio de Janeiro, Brazil, W. Frakes (Ed.), IEEE Computer Society Press, Los Alamitos, California, 1994, pp. 74-83

[Gog83] J. Goguen. LIL – A Library Interconnection Language. In Report on Program Libraries Workshop, SRI International, Menlo Park, California, October 1983, pp. 12-51

[Gom92] H. Gomaa. An Object-Oriented Domain Analysis and Modeling Method for Software Reuse. In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January 1992

[Gon81] L. Gonzales. A domain language for processing standardized tests. Master' s thesis, Department of Information and Computer Science, University of California, 1981

[GT96] J. Goguen and W. Tracz. An Implementation-Oriented Semantics for Module Composition. Draft available from [ADAGE], 1996

[Hay94] 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, available from [ADAGE]

[HNC+90] J. Hess, W. Novak, P. Carroll, S. Cohen, R. Holibaugh, K. Kang, and A. Peterson. A Domain Analysis Bibliography. Technical Report, CMU/SEI-90-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990. Reprinted in [PA91], pp. 258- 259. Also available from http://www.sei.cmu.edu

[Hol93] R. Holibaugh. Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA). Version 1.3, Technical Report, CMU/SEI-92-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1993, http://www.sei.cmu.edu

[HT94] F. Hayes-Roth and W. Tracz. DSSA Tool Requirements For Key Process Functions. Version 2.0, Technical Report, ADAGE-IBM-93-13B, October 24, 1994, available from [ADAGE]

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

[Kru93] R. Krut. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report, CMU/SEI-93-TR-11, ESC-TR-93-188, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993, http://www.sei.cmu.edu

[Kic97] G. Kiczales. Verbal Excerpt from the ECOOP'97 tutorial on "Designing High-Performance Reusable Code", Jyväskylä, Finland, 1997

[LEW96] J. Loeckx, H. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley & Teubner, 1996

[LHK87] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *Anna: A Language For Annotating Ada Programms. Language Reference Manual.* Lecture Notes in Computer Science, no. 260, Springer-Verlag, 1987

[LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Transaction on Software Engineering*, vol. 21, no. 4, April 1995, pp. 336-355

[Lub91] M. Lubars. Domain Analysis and Domain Engineering in IDeA. In [PA91], pp. 163-178

[MBSE97] Software Engineering Institute. Model-Based Software Engineering. WWW pages, URL: http://www.sei.cmu.edu/technology/mbse/, 1997 (viewed)

[MC96] T. Moran and J. Carroll, (Eds.). *Design Rationale: Concepts, techniques, and use*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1996

[Nei80] J. Neighbors. Software construction using components. Ph.D. dissertation, (Tech. Rep. TR-160), Department Information and Computer Science, University of California, Irvine, 1980

[Nei84] J. Neighbors. The Draco Approach to Construction Software from Reusable Components. In *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 564-573

[Nei89] J. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In [BP89], pp. 295-319

[OPB92] E. Ostertag, R. Prieto-Díaz, and C. Braun. Computing Similarity in a Reuse Library System: An AI-Based Approach. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 3, July 1992, pp. 205-228

[PA91] R. Prieto-Diaz and G. Arrango (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991

[Par76] D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1-9

[PC91] S. Peterson and S. Cohen. A Context Analysis of the Movement Control Domain for the Army Tactical Command and Control System. Technical Report, CMU/SEI-91-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991

[PF87] R. Prieto-Díaz and P. Freeman. Classifying Software for Reusability. In *IEEE Software*, January 1987, pp. 6-16

[Pri85] R. Prieto-Díaz. A Software Classification Scheme. Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine, 1985

[Pri87] R. Prieto-Díaz. Domain Analysis For Reusability. In Proceedings of COMPSAC' 87, 1987, pp. 23-29 and reprinted in [PA91], pp. 63-69

[Pri91a] R. Prieto-Díaz. Implementing Faceted Classification for Software Reuse. In *Communications of the ACM*, vol. 34, no. 5, May 1991, pp. 88-97

[Pri91b] R. Prieto-Díaz. Reuse Library Process Model. Technical Report, IBM STARS 03041-002, Electronic Systems Division, Air Force Systems Command, USAF, Hanscom Air Force Base, Hanscom, Massachusetts, July, 1991

[SC93] M. Simos and R.E. Creps. Organization Domain Modeling (ODM), Vol. I - Conceptual Foundations, Process and Workproduct Descriptions. Version 0.5, Unisys STARS Technical Report No. STARS-UC-05156/024/00, STARS Technology Center, Arlington, Virginia, 1993

[SCK+96] 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, http://www.organon.com

[Sha77] D. Shapere. Scientific Theories and Their Domains. In *The Structure of Scientific Theories*, F. Suppe (Ed.), University of Illinois Press, 1977, pp. 519-565

[Shu91] S. Shum, Cognitive Dimensions of Design Rationale. In *People and Computers VI: Proceedings of HCI'91*, D. Diaper and N. Hammond, (Eds.), Cambridge University Press, Cambridge, 1991, pp. 331-344, http://kmi.open.ac.uk/~simonb/DR.html

[Sim91] M. Simos. The Growing of an Organon: A Hybrid Knowledge-Based Technology for Software Reuse. In [PA91], pp. 204-221

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice-Hall, 1996

[SJ95] Y. Srinivas and R. Jüllig. Specware™: Formal Support for Composing Software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, (Ed.), Lecture Notes in Computer Science, vol. 947, Springer-Verlag, Berlin, 1995, also http://www.kestrel.edu/

[Smi90] D. Smith. KIDS: A Semiautomatic Program Development System. In *IEEE Transactions on Software Engineering*, vol. 16, no. 9, September 1990, pp. 1024-1043

[Smi96] D. Smith. Toward a Classification Approach to Design. In *Proceedings of Algebraic Methodology & Software Technology, AMAST' 96*Munich, Germany, July 1996, M. Wirsing and M. Nivat (Eds.), LCNS 1101, Springer, 1996, pp. 62-84

[SPC92] Software Productivity Consortium. Reuse Adoption Guidebook. Technical Report, SPC-92051-CMC, Software Productivity Consortium, Herndon, Virginia, 1992, http://www.asset.com

[SPC93] Software Productivity Consortium. Reuse-Driven Software Processes Guidebook. Version 02.00.03, Technical Report, SPC-92019-CMC, Software Productivity Consortium, Herndon, Virginia, November 1993, http://www.asset.com

[SPW95] D. Smith, E. Parra, and S. Westfold. Synthesis of High-Performance Transportation Schedulers. Technical Report, KES.U.1, Kestrel Institute, February 1995, http://www.kestrel.edu/

[Sri91] Y. Srinivas. Algebraic specification for domains. In [PA91], pp. 90-124

[STARS94] Software Technology For Adaptable, Reliable Systems (STARS). Army STARS Demonstration Project Experience Report. STARS Informal Technical Report, STARS-VC-A011R/002/01, November 30, 1994

[Sun83] S. Sundfor. Draco domain analysis for real time application: The analysis. Technical Report, RTP 015, Department of Information and Computer Science, University of California, Irvine, 1983

[TC92] W. Tracz and L. Coglianese. DSSA Engineering Process Guidelines. Technical Report, ADAGE-IBM-9202, IBM Federal Systems Company, December 1992

[TH93] D. Tansley and C. Hayball. *Knowledge-Based Systems Analysis an Design: A KADS Developer's Handbook*. Prentice Hall, 1993

[THE+94] A. Terry, F. Hayes-Roth, L. Erman, N. Coleman, M. Devito, G. Papanagopoulos, B. Hayes-Roth. Overview of Teknowledge's Domain-Specific Software Architecture Program. In *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 4, October 1994, pp. 68-76, see http://www.teknowledge.com/DSSA/

[Tra93] W. Tracz. Parameterized programming in LILEANA. In *Proceedings of ACM Symposium on Applied Computing, SAC'93*, February 1993, pp. 77-86

[Tra95] W. Tracz. Domain-Specific Software Architecture Pedagogical Example. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995, pp. 49-62, also available from [ADAGE]]

[TTC95] R. Taylor, W. Tracz, and L. Coglianese. Software Development Using Domain-Specific Software Architectures: CDRL A011 – A Curriculum Module in the SEI Style. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 5, December 1995, pp. 27-37, also available from [ADAGE]

[UML97a] Rational Software Corporation. UML (Unified Modeling Language) Glossary. Version 1.0 1, 1997, http://www.rational.com

[Uni88] Unisys. Reusability Library Framework AdaKNET and AdaTAU Design Report. Technical Report, PAO D4705-CV-880601-1, Unisys Defense Systems, System Development Group, Paoli, Pennsylvania, 1988

[VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro. FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. In [DP98], pp. 166-175, see http://www.intecs.it

[VG90] W. Vitaletti and E. Guerrieri. Domain Analysis within the ISEC Rapid Center. In *Proceedings of Eighth Annual National Conference on Ada Technology*, March 1990

[Wei96] D. Weiss. Creating Domain-Specific Languages: The FAST Process. Transparencies presented at The first ACM-SIGPLAN Workshop on Domain-Specific Languages, Paris, France, January 18, 1997, http://www-sal.cs.uiuc.edu/~kamin/dsl/index.html

[Wit94] J. Withey. Implementing Model Based Software Engineering in your Organization: An Approach to Domain Engineering. Draft, Technical Report, CMU/SEI-94-TR-01, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994

[Wit96] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1996, http://www.sei.cmu.edu

[WP92] S. Wartik and R. Prieto-Díaz. Criteria for Comparing Domain Analysis Approaches. In *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 3, September 1992, pp. 403-431

[Zal96] N. Zalman. Making The Method Fit: An Industrial Experience in Adopting Feature-Oriented Domain Analysis (FODA). In *Proceedings of the Fourth International Conference on Software Reuse*, M. Sitaraman, (Ed.), IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 233-235