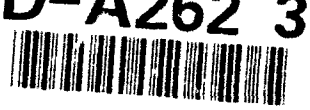


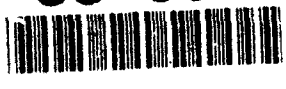
CRITERIA FOR COMPARING DOMAIN ANALYSIS APPROACHES

MDA 972-92-J-1018

SPC-92117-CMC

VERSION 01.00.00

DECEMBER 1991



CRITERIA FOR COMPARING DOMAIN ANALYSIS APPROACHES

SPC-92117-CMC

VERSION 01.00.00

DECEMBER 1991

Steven Wartik
Rubén Prieto-Díaz

Reprinted for the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

February 1993

SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

UNIX is a registered trademark of UNIX System Laboratories, Inc.

CONTENTS

FIGURES

TABLES

This page intentionally left blank.

ACKNOWLEDGEMENTS

Thanks are due to Grady Campbell for providing the original criteria for comparison and to Jim O'Connor for some thoughtful analysis of the differences between the various processes and products. The document's present state is the result of insightful reviews by Grady Campbell, Bill Frakes, Fred Hills, Rich McCabe, Sam Redwine, and Dave Weiss.

This page intentionally left blank.

EXECUTIVE SUMMARY

Several of the Software Productivity Consortium's key technologies for process and reuse improvement incorporate domain analysis. However, the Consortium has three approaches to domain analysis associated with it, and these approaches differ in goals, end products, and processes. Lacking criteria for comparing the approaches, no one could justify using one approach over another. This has been something of a problem for Consortium projects looking to select an approach. Moreover, two of the approaches are still maturing; their architects need to know their strengths, weaknesses, and applicabilities if they are to become more useful. Member companies, who are increasingly interested in performing domain analysis, also need to know which approach to use.

This report describes criteria for comparing domain analysis approaches. An organization can use them to determine if a domain analysis approach will meet their needs. To some degree, they also rank approaches in order of desirability.

Domain analysis occurs in response to some need, so the Consortium looked for software development factors that dictate the suitability of a domain analysis approach for a given organization. It settled on the following five:

1. Software process needs: Constraints on the process model used to develop software.
2. Existing software base: The number and availability of existing applications in a domain and their characteristics.
3. Business objectives: The long-term and short-term plans for building and using the products that result from domain analysis.
4. State of domain knowledge: The maturity of the domain.
5. Intended use of information repositories: How software developers are to use the products of domain analysis.

These factors characterize an organization. By contrast, the comparison criteria say nothing about an organization. They only characterize differences among domain analysis approaches. However, the Consortium studied the relationships between those differences and organizational factors, and it created an informal mapping between the two. An organization therefore uses the criteria by first characterizing itself in terms of the five factors. It then characterizes a domain analysis approach in terms of the criteria and, using the mapping, determines if the approach satisfies its needs.

The Consortium derived the criteria by studying similarities and differences among existing domain analysis approaches. Similarities included high-level objectives (the creation of artifacts that allow for effective reuse and the capture and formalization of domain knowledge), the sources of domain knowledge

(domain experts, reference materials, existing systems), and—to the extent that their objectives are similar—agreement on difficulties in performing domain analysis (the need for precise definitions of domain artifacts, how to validate the results of domain analysis, and economic considerations).

The Consortium uncovered ten criteria by which researchers or practitioners can contrast the approaches. Table 1 summarizes them.

Table 1. Summary of Comparison Criteria

Criterion	Meaning	Choices
Definition of "domain"	What a domain encompasses, how that influences what is considered a domain, and how organizations satisfy business goals accordingly.	• Application area • Business area
Determination of problems in the domain	The approach used to arrive at the set of problems that make up a domain.	• Problem-oriented • Solution-oriented • Problem-/solution-oriented
Permanence of domain analysis results	Whether products of domain analysis evolve.	• Permanent • Mutable
Relation to the software development process	How domain analysis activities fit into a software process, model activities (or vice versa).	• Pre-requirements, dependent • Pre-requirements, independent • Meta-process
Focus of analysis	The fundamental concept on which analysts focus during analysis.	• Objects and operations • Decisions
Paradigm of problem space models	The fundamental concept emphasized by the problem space model that the analysts derive.	• Generic requirements • Decision model • Both
Purpose and nature of domain models	Intended uses of the products of domain analysis.	• Repository • Software specification • Process specification
Organizational models of domains and projects	Possible organizations a company might use to maximize the potential of domain analysis.	• Circumstance-driven • Project-driven • Domain-driven
Approach to reuse	Strategies for exploiting the reusable components generated during domain analysis and implementation.	• Opportunistic • Systematic
Primary product of domain development	Most significant product resulting from domain implementation, guiding how other products will be used.	• Reuse library • Application engineering process

The Consortium applied the criteria to six approaches—the three Consortium approaches and three developed elsewhere—so it could see trends in domain analysis. It discovered that approaches may share similar goals but can meet these goals in very different ways. It also found that, according to the criteria, no two approaches are exactly alike. This indicates that the criteria discriminate appropriately.

When the study began, the Consortium hoped that one outcome would be a recommendation for how to create a unified approach to domain analysis. Such an approach would fill all Consortium needs and would be useful in most situations. After performing this study, the Consortium does not believe a unified approach is useful. Domain analysis occurs in response to some (software) need. There are many different types of needs, and the products to support them vary. A unified approach would be overly complex. The Consortium should instead supply a decision process for selecting among domain analysis approaches. This report is a first step toward such a decision process. Projects that require domain analysis could then define their needs precisely and select the approach that best meets those needs. Meanwhile, domain analysis researchers must continue to improve their approaches and to clarify the niche they fill.

This page intentionally left blank.

1. INTRODUCTION

This report presents a framework for comparing domain analysis approaches. The Consortium has several goals in doing so:

1. To provide a means for practitioners to determine which approach best suits their needs.
2. To study the feasibility of a unified approach to domain analysis that is applicable across domains.
3. To show trends in domain analysis research.
4. To provide domain analysis researchers with a common conceptual ground.

The Consortium based the comparison on criteria derived from six existing approaches. The criteria show conceptual differences that relate to contextual needs, i.e., they determine how and where practitioners should use a given domain analysis process.

The remainder of this section states the problem more exactly and defines some concepts.

1.1 PROBLEM STATEMENT

One characteristic of an emerging technology is the many manifestations it assumes before its practitioners recognize standard terminology and semantics. This is certainly true of domain analysis which, in its relatively brief existence, has come to mean many things to many people. Surveys of domain analysis approaches (e.g., [Prieto-Díaz and Arango 1991]) show underlying similarities; broadly speaking, everyone agrees that domain analysis is the analysis of some area, leading toward some predetermined goal. However, the diverse goals, products, and processes of the approaches—which this report will cover in depth—have left people confused about which approach will best fit their needs.

This confusion stems from two causes. One is the conflicting goals people have for domain analysis. Although originally intended to promote software reuse (Neighbors 1984), domain analysis is also useful for capturing domain knowledge (Shlaer and Mellor 1989), for helping people learn about domains (Arango 1988), or for combinations of these goals. A second cause of confusion is the difference between the products that result from various approaches. How, for example, should a practitioner determine the circumstances under which the faceted classification scheme espoused by Prieto-Díaz (Prieto-Díaz 1987) characterizes differences between components more accurately than the hierarchical structures of FODA (feature-oriented domain analysis) (Kang et al. 1990)? Lubars represents abstract designs using an extended flowchart model incorporating object-oriented concepts (Lubars 1991). How does this compare to Synthesis (Software Productivity Consortium 1991), which does not prescribe a particular model for abstract designs, or to FODA, which uses the Design Approach for Real-Time Systems (DARTS) design method (Gomaa 1984)?

Domain analysis is still immature. Eventually, one approach may become standard, but this seems unlikely. People are already applying domain analysis on a broad spectrum of projects. The diversity of these projects introduces external factors that influence which domain analysis approach is appropriate. Sorting out these factors introduces more confusion than the two causes listed above; indeed, it drives those causes. FODA abstract designs probably will be more useful to a project that intends to use DARTS than to one using object-oriented design. The opposite holds true for Lubars' approach.

If no single approach meets all situations, then people must understand how to select the one that best fits their needs. The person making such a selection must understand how domain analysis can fit into a software development process. He must also be able to evaluate the possible benefits of using domain analysis at each step of the software process his company employs and, more particularly, he must be able to compare the short-term and long-term benefits and costs provided by the approaches he is considering. He must therefore be prepared to weigh the immediate needs and possibilities of his organization against the expected future gain. In short, he faces a challenging task.

1.2 DOMAIN ANALYSIS AND SOFTWARE REUSE

Domain analysis has other uses besides reuse, but most people want to use it to that end. The Consortium therefore gives a general statement of how it believes domain analysis affects reuse. The definitions below apply to all domain analysis approaches it has studied.

Researchers have identified different types of reuse. They categorize the types based on various factors (see [Basili et al. 1987]), such as type of entity being reused (code, design, documentation, etc.) or the kinds of adaptation needed (verbatim reuse, reuse of generic parts, reuse of fragments, etc.). Practitioners are asking for domain analysis approaches that accommodate all types of reuse.

Domain analysis is concerned with knowledge acquisition and with methods to make use of that knowledge. Software reuse involves using these methods. Therefore, the Consortium is interested in what these methods can offer. Assume that a developer has a stated need—a module specification, for instance—and hopes to fulfill it by reusing an existing component rather than writing one from scratch. The base situation (no domain analysis) implies no defined methods for reuse. A minimal domain analysis approach must yield methods that tell developers what opportunities they have for reuse—that is, what components are, or need to be, available. The ideal domain analysis approach would define methods that tell the developer everything he needs to know about reuse: which component best matches the specification, how to adapt it if it does not meet the specification exactly, and so on. The approach thus mechanizes reuse.

This suggests that researchers classify types of reuse in terms of software development processes, for two reasons. First, they can define a scale based on just how mechanical the methods are. The more mechanical the method, the less need for creativity, which implies less effort. Second, domain analysis is not limited to code reuse, and methods for code reuse are not likely to be productive during software requirements production. The ideal approach to domain analysis must define appropriate methods for all relevant software process activities.

Based on this logic, the Consortium has defined three categories of reuse regarding the sophistication of methods for responding to a reuse need. The definitions are based on the methods that developers have available to help them reuse software. The categories are:

- If the software process specifies no methods, developers must practice **ad hoc reuse**. This is equivalent to informal, superficial domain analysis.

- If the software process specifies methods to identify what types of components developers may reuse at a given time, and perhaps how to find such components in a set of existing software assets (such as a reuse library), developers practice **opportunistic reuse**. In other words, their opportunities for reuse are predefined, but not how they take advantage of those opportunities.

- If the software process specifies methods that define what components to reuse and how to adapt them, developers practice **systematic reuse**. That is, the opportunities are predefined, and a process for making use of those opportunities is specified.

Indexing schemes and searching heuristics are examples of methods used in opportunistic reuse (Frakes and Gandel 1987). Application generators illustrate the types of methods developers used in systematic reuse. For instance, a parser-generator implements systematic reuse for the domain of context-free languages. Given a reuse need (stated as a grammar), it uses predefined algorithms (methods) to build a parser. The usual approach is to adapt a canonical parser to recognize the grammar. (However, developers can systematically adapt parts, too.)

Such tools are currently available in relatively few domains. This reflects the constant maturation of technology—domain analysis technology, but also of technology in general. Researchers have only recently begun thinking about defining methods to promote reuse. Meanwhile, new discoveries continually create new domains that practitioners must analyze. There is a backlog of domains to analyze. Even the practice of opportunistic reuse is limited by the need to analyze a domain to the point where practitioners understand it enough to categorize components meaningfully.

Therefore, all types of reuse play a role in software development. No one type is clearly superior to another; the one that is most appropriate depends on the software being built. The goal of domain analysis is to produce the best products and methods for a domain. Which domain analysis approach is right for the domain therefore depends on the type of reuse that application developers practice.

1.3 DOMAIN ANALYSIS IN CONTEXT

The previous section leads to an important point: comparing domain analysis approaches requires considering the context in which practitioners will use an approach. The questions posed in Section 1.1 are only meaningful within such a context. Domain analysis is a means to an end, and that end imposes constraints.

This report has already shown how the type of reuse influences the approach an organization chooses. Section 4 gives criteria for comparing approaches. All are related to contextual factors that an organization using domain analysis must consider, in terms of its needs, before selecting an approach.

This section discusses such contextual factors. The list is not exhaustive, although it represents a useful selection. As the Consortium gains more experience with domain analysis, it will no doubt understand context issues more fully than it does today.

Domain analysis approaches are considered in the context of the following five factors:

1. ***Software Process Needs.*** Some domain analysis techniques are intended for a certain process model (e.g., waterfall). Others require instituting specific process models, sometimes nonstandard, that may or may not fit into a given organizational structure. For cost reasons, or for contractual purposes, process needs may mandate or obviate an approach. In addition,

if an organization wants or needs software process maturity (Humphrey 1989), it should consider the influence of domain analysis.

2. ***Existing Software Base.*** Successful domain analysis hinges on the ability of domain experts to crystallize their knowledge. They often do so by examining properties of existing applications in the domain. They therefore need access to such applications. Moreover, the products for domain analysis generally include a library of reusable code components, or at least a specification for one. More existing applications potentially means a richer library.

   Domain analysts must also consider certain properties of the existing software. Most domain analysis approaches are not concerned simply with existing properties of a domain. They attempt to account for how the domain will evolve as new applications and the effects of technology introduce the unforeseen.

3. ***Business Objectives.*** An organization should be cognizant of how it intends to use the products of domain analysis. The analysis processes are potentially costly. The organization must understand this, have a model for absorbing the costs, and know when to expect to recover these costs. The organization must understand both its short-term and long-term needs. This is especially important in planning domain evolution (if evolution is a goal). The organization must know its current and future customer needs and have a plan prioritizing domain analysis and implementation efforts to meet those needs.

4. ***State of Domain Knowledge.*** The more mature (and, consequently, usually less volatile) a domain, the easier domain analysis will be. Conversely, there is little point in an organization undertaking a multi-year, extensive analysis for a domain whose properties it understands poorly—the benefits will be minimal in comparison to the costs. However, certain aspects of domain analysis are still quite viable in an immature domain. An organization should understand what those aspects are—planning for domain evolution, for example—and choose an approach that does not stress others.

5. ***Intended Use of Information Repositories.*** Once the domain analysts specify and implement domain-specific products, they make the products generally available to application developers. The developers can use the information in these products in many ways. One, of course, is as reusable software components, but domain analysis researchers have proposed many other possibilities. They range from a general knowledge base to creation of assessment and simulation tools. Some domain analysis processes yield abstract designs but no code, whereas others yield code but no designs. Thus an organization must determine just what types of products it would like to obtain from domain analysis and how it wants to use those products.

1.4 ORGANIZATION OF THIS REPORT

This report is organized as follows:

- Section 2 gives background information on six domain analysis approaches, three from the Consortium and three developed by other researchers.
- Section 3 covers characteristics shared by all domain analysis approaches.
- Section 4 presents the comparison criteria.

- Section 5 applies these criteria to the six approaches, showing commonalities and diversity among them.
- Section 6 presents the Consortium's conclusions.

1.5 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

Serif font ..... General presentation of information.

Italicized serif font ..... Publication titles.

Boldfaced serif font ..... Section headings and emphasis.

This page intentionally left blank.

2. AN OVERVIEW OF SOME DOMAIN ANALYSIS APPROACHES

As Section 1 hinted, the Consortium has drawn much of the material in this report from studying three domain analysis approaches in use at the Consortium and from three others created by industrial and academic researchers. This section sets the stage for the remainder of the report by presenting, for each approach, an overview of its history and its important concepts, a description of its process and products, and examples of domains to which it has been applied.

A little history may help in understanding these descriptions. This report grew from a need to clarify domain analysis goals and processes at the Consortium. There were three separate domain analysis approaches because three researchers, each with different views on domain analysis, had worked there. The Consortium's management recognized the need to avert a problem and ordered a study of the three approaches. The goals were to determine if one of the three approaches was best, or at least to characterize the circumstances under which each approach was advantageous. The Consortium initially had hopes of synthesizing its findings to create a single, unified domain analysis approach that it could recommend be used in almost all situations.

Figure 1 shows a genealogy of domain analysis approaches. While it is not complete, it captures the most important paths that domain analysis researchers have followed. Arrows indicate influence. For example, information hiding concepts and Coad's approach to domain analysis (Coad 1989) influenced Jaworski's approach (Jaworski et al. 1990). Object-oriented analysis (OOA) and Neighbors' pioneering work on domain analysis influenced Coad's approach.

Figure 1 includes all three Consortium approaches (Jaworski, Prieto-Díaz, and Synthesis). They do not cover all branches, however, indicating that a comparison of domain analysis approaches based only on those three might miss important contributions of other researchers. The Consortium therefore included other approaches. It first added FODA to the study, and later KAPTUR (Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales) and Lubars' approach. Because Shrinivas' interests were theoretical—he concentrated on a particular aspect of domain modeling and did not define a complete domain analysis process—it excluded the branch of the genealogy that includes his work. The six approaches this report treats are in boldface type in Figure 1.

2.1 JAWORSKI'S APPROACH

2.1.1 Overview

A group at the Consortium under the direction of Dr. Alan Jaworski created the first of the three Consortium approaches covered here. The fundamental concept underlying the approach is the use of Coad's OOA techniques (Coad 1989) to yield the entities that comprise a domain. Thus a domain is a group of interacting objects and operations. These objects and operations are requirements-level; that is, they reflect domain concepts, not implementation details.

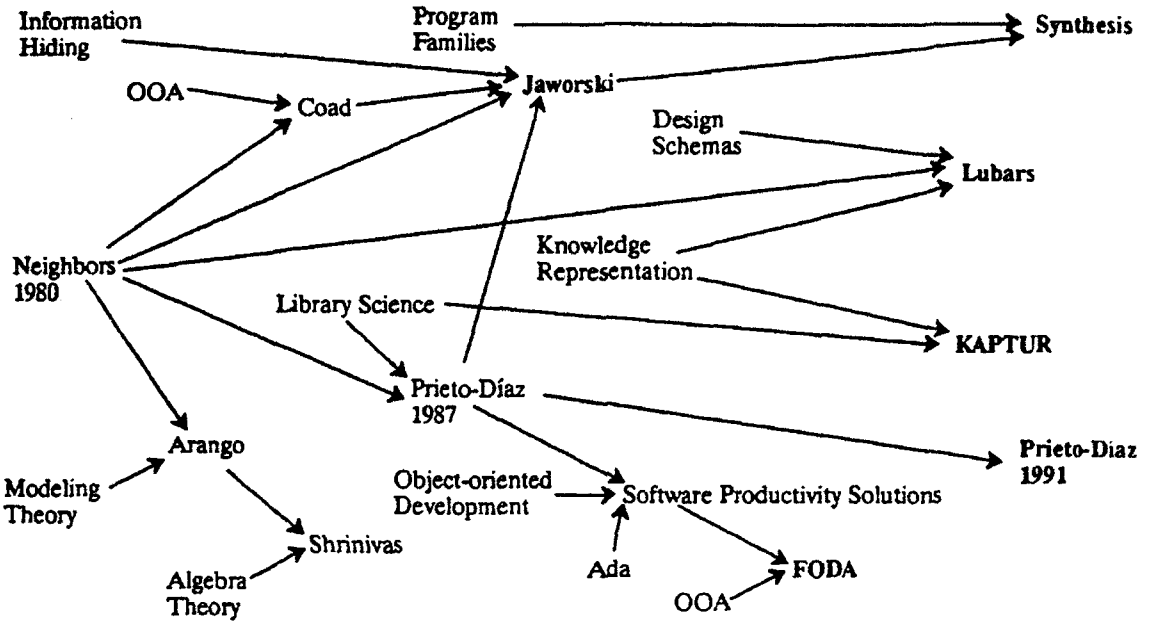


Figure 1. Genealogy of Domain Analysis Approaches

2.1.2 Process and Products

Jaworski defines a four-step domain analysis process, shown in Figure 2, which results in the following outputs:*

- A domain definition, which serves as an informal (but careful) statement of what is and is not part of the domain. It provides a working specification for subsequent products, especially the canonical requirements, and it helps application implementors determine if the domain contains components that meet their needs.
- A feasibility analysis that shows the cost-effectiveness of implementing reusable components in the domain (Cruickshank and Gaffney 1991). Organizations use this product to determine whether the domain is economically viable, and if so, to prioritize components.
- A knowledge base containing information about the domain—ideally, anything deemed relevant, from laws governing the domain to reusable artifacts. Domain developers use the knowledge base as they implement operations in the domain. Application developers use it to understand more about the domain.
- Canonical requirements, which are black-box statements of capabilities and constraints, and of the variabilities that distinguish instances. They are expressed as objects, and so are specifications of reusable components. During the "domain implementation" activity, which follows domain analysis, domain developers design and implement these components. Application developers, who, having recognized that a canonical requirement matches a reuse need, use the associated variabilities to tailor the requirement to their exact need. They can also tailor the corresponding reusable component; the result, then, is an instance of a requirement and a corresponding design and implementation. Canonical requirements are often termed "generic requirements"; this report uses the latter term.

* To aid comparison, the figures in this section do not always use the same activity names or presentation format as in the papers from which they are drawn.

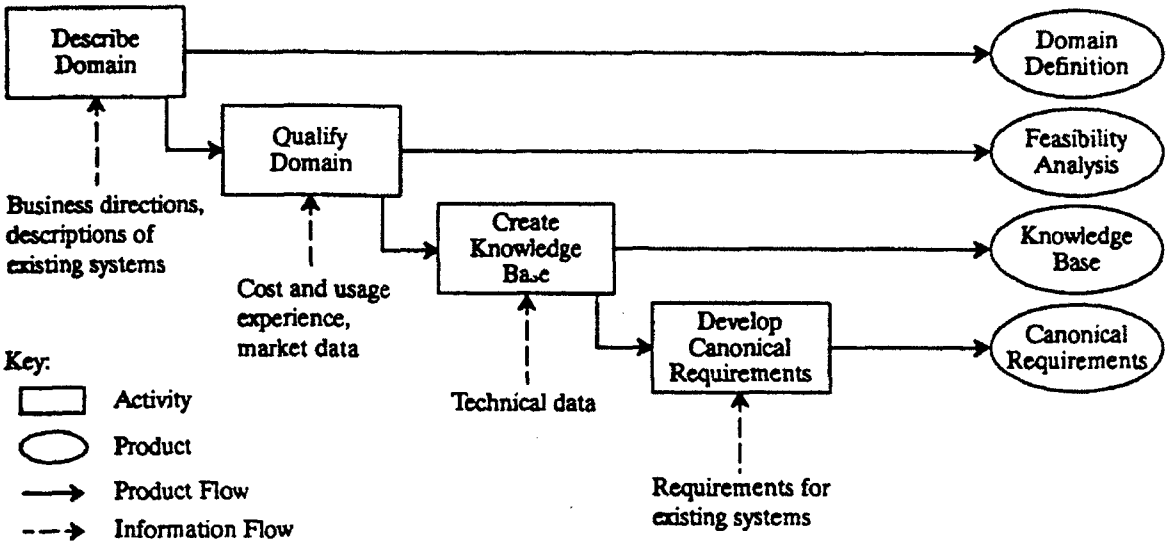


Figure 2. Jaworski's Process for Domain Analysis

Domain analysis and domain implementation together comprise "domain engineering," a process of defining and implementing reusable components. Application developers use the results of domain engineering in "application engineering," the process for implementing instances of applications within a domain. Jaworski does not explicitly define this application engineering process, although he suggests a waterfall-like model.

2.1.3 Examples

The Consortium has tried, experimentally, on several domains, including the Satellite Operations Control Center (SOCC) domain, the domains of job management systems (Snodgrass et al. 1990), and automobile cruise controls.

2.2 DOMAIN ANALYSIS IN SYNTHESIS

2.2.1 Overview

The Synthesis domain analysis process (Campbell et al. 1991) is in some respects a refinement of Jaworski's approach, although it is oriented toward program families rather than OOA. The concepts developed by the Software Cost Reduction project at the Naval Research Lab (Parnas, Clements, and Weiss 1985) and subsequent work on the Spectrum application development environment from Software Architecture and Engineering, Inc. have heavily influenced it. The fundamental concepts underlying the work are:

- The knowledge derivable from a domain analysis effort is sufficient to design a process for engineering applications in that domain. An application can be engineered in terms of domain concepts rather than software design or programming language concepts. This means that system building reduces to resolving requirements and engineering decisions representing variations within a domain.
- Program family concepts (Parnas 1976) apply to domain specifications. In other words, a domain is a family of applications sharing many common features but also varying in precisely defined ways.

- A domain analyst can create a mapping from differences among applications in a domain to a family of designs. Application developers can use this mapping to adapt the design mechanically.

2.2.2 Process and Products

Figures 3 and 4 show the activities in the Synthesis domain analysis process. Figure 3 shows only domain analysis activities, and in a form intended for comparison with the other approaches in this section (note that it omits domain verification). Figure 4 shows the Synthesis domain analysis process in the context of a domain engineering process. In Synthesis, domain engineering produces a process and environment for engineering applications within a domain. Domain analysis produces what amounts to a specification of that environment, including:

- A domain definition which, in effect, combines Jaworski's domain definition and feasibility analysis.

- A domain specification, a specification of the domain precise enough to facilitate domain implementation. The domain specification contains:

  - A decision model, describing decisions application developers must make to identify a specific application within a domain.

  - Product requirements, specifying shared and unique behavior across all applications within a domain.

  - Process requirements, defining how application developers can systematically specify, generate, assess, and validate an application in terms of the decision model.

  - Product design, specifying the architecture of each work product, the relationship between the decision model and the work products, and adaptable components to be used to create the work products.

In the domain implementation phase of domain engineering, domain developers use the domain specification to build reusable components and the environment in which application developers will build applications. The domain developers verify these products using the domain specification.

Figure 4 shows that the product of domain implementation is application engineering process support. What this means is that Synthesis defines the process for implementing instances of the domain in application engineering (namely, the process requirements). This process is termed the "application engineering process." Application developers, guided by the process, use the decision model to identify the application that meets their needs. Software development becomes a concurrent interaction between a domain engineering process and a set of application development processes. Domain engineering provides a process to build applications; that process in turn provides feedback on the products of domain engineering (see Figure 5). A "domain management" activity, responsible for defining how the domain should evolve to meet short-term application needs and long-term strategic goals, guides domain engineering.

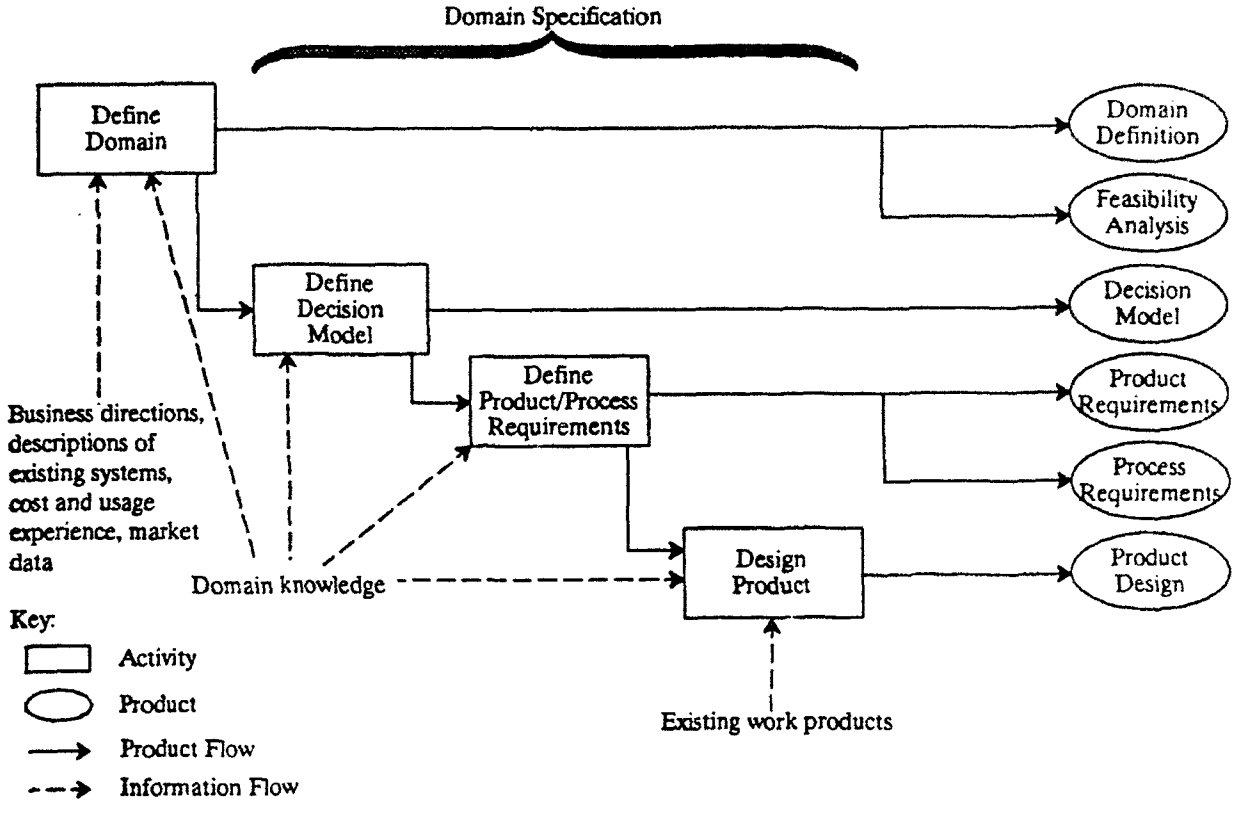


Figure 3. Synthesis Domain Analysis Process (Partial)

2.2.3 Examples

The development of the Spectrum environment informally followed this approach. The Consortium has tried it experimentally on a variety of domains, including the Host-At-Sea Buoy system (Weiss 1990), job management systems (Snodgrass et al. 1990), design composers (Burkhard et al. 1990), and air traffic display/collision warning monitors (Campbell et al. 1991). A pilot project in the domain of communication control systems, in cooperation with Rockwell International, is currently trying it.

2.3 THE FACETED CLASSIFICATION APPROACH OF RUBÉN PRIETO-DÍAZ

2.3.1 Overview

Methods for deriving classification schemes in library science and on methods for systems analysis are the basis of the approach by Rubén Prieto-Díaz (Prieto-Díaz 1987; Prieto-Díaz 1990; Prieto-Díaz 1991). The process is a "sandwich" approach: the classification process supports bottom-up activities, and systems analysis supports top-down activities.

The view of domain analysis in this methodology follows the line of thought pioneered in Neighbors' DRACO system "to identify objects and operations for a class of similar systems" (Neighbors 1984). A primary original objective, then, was making that information readily available. Neighbors indicated the importance of domain analysis in reusability but did not address how to do it. Prieto-Díaz's was the first proposed methodology to do domain analysis for reusability.

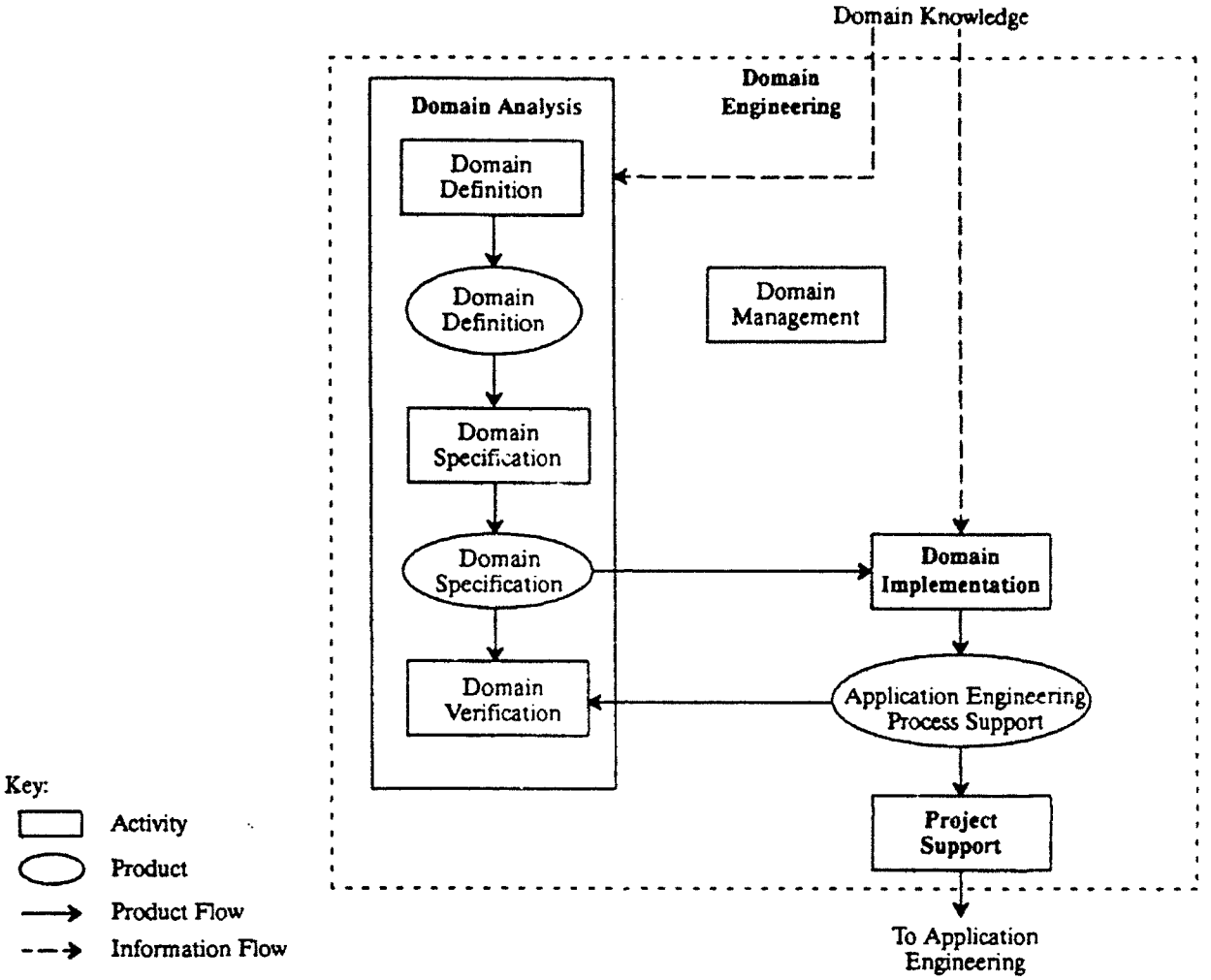


Figure 4. Synthesis Domain Engineering Process

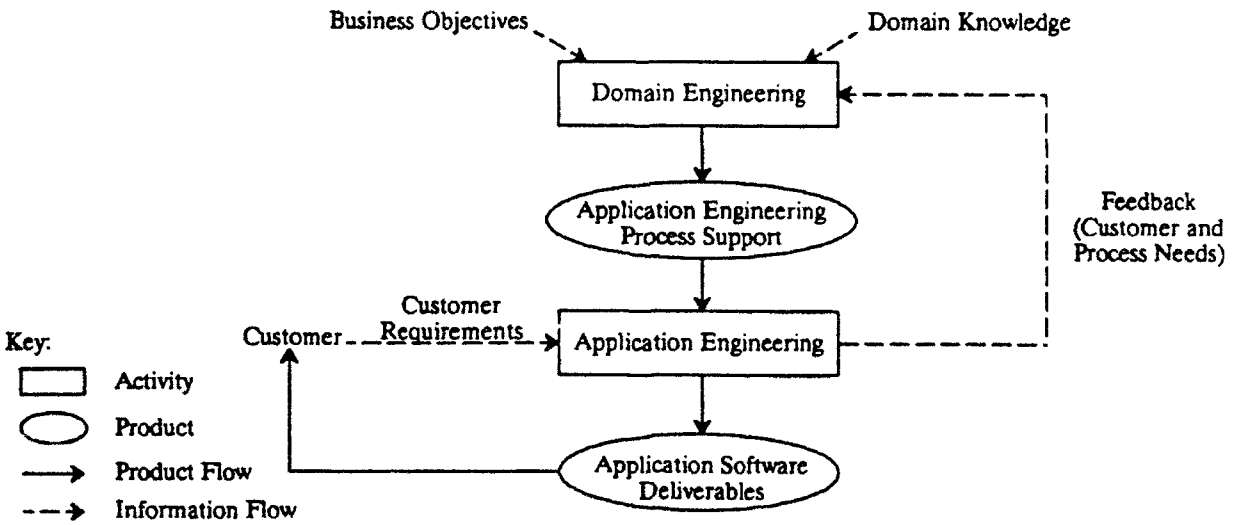


Figure 5. Software Development in Synthesis

2.3.2 Process and Products

Prieto-Díaz developed this approach in two stages. In its first version (Prieto-Díaz 1987), the three top-level activities shown in Figure 6 describe the process. The second activity, analyzing the domain, is the core activity. It consists of multiple iterations of selection, abstraction, and classification of functions, objects, and relationships in much the same way that librarians derive specialized classification schemes. The approach proved successful in identifying generic objects and operations in a domain but was weak in supporting the selection and encapsulation of reusable components.

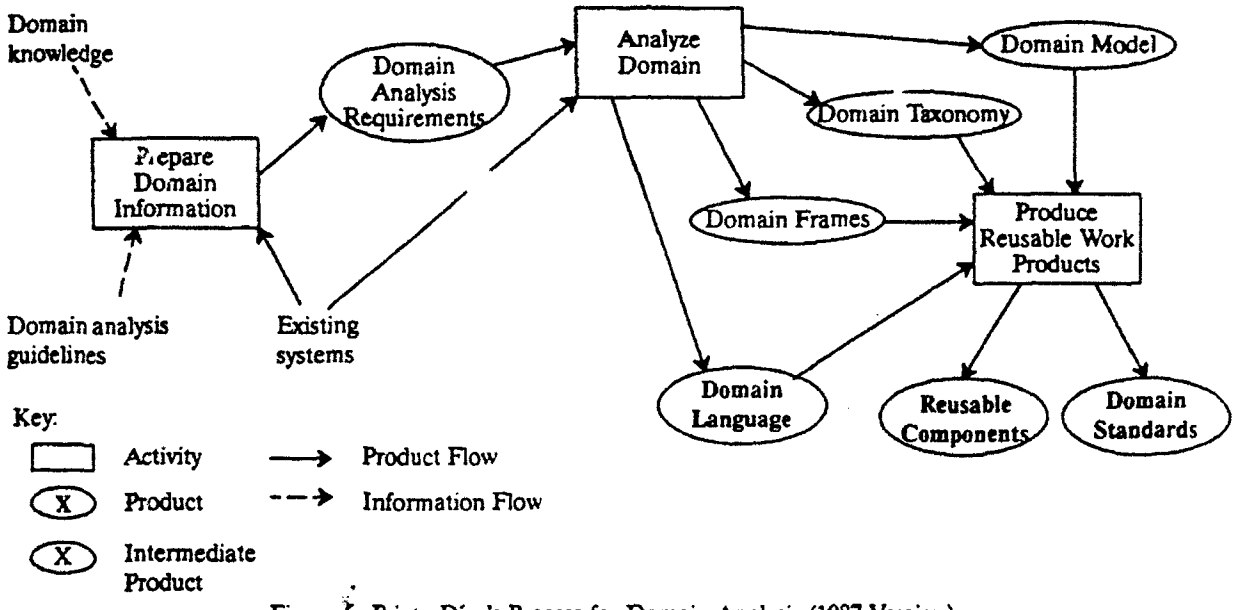


Figure 6. Prieto-Díaz's Process for Domain Analysis (1987 Version)

The second version (Prieto-Díaz 1991), shown in Figure 7, is broader and more cohesive. It proposes a framework to integrate domain analysis in a software development process. In this framework the domain analyst continually reviews and refines the products of domain analysis as practitioners construct new systems in the domain. The new version of the methodology complements the first version's bottom-up approach of identifying objects and operations with a top-down, systematic analysis to identifying domain models.

During top-down analysis, the domain analyst analyzes high-level designs and requirements of current and new systems for commonality. Two activities support top-down analysis. The first, preparing domain information, yields what is, in effect, a preliminary domain analysis. It consists of:

- The domain definition, an informal statement of the types of applications in the domain.
- The basic domain architecture, a high-level description of architectural properties shared by applications in the domain. It contains the following information:
  - A canonical structure common to all systems in the domain. It provides guidance to the bottom-up analysis by identifying key components common to domain applications.
  - Identification of stable and variable characteristics. These characteristics complement the canonical structure and support the selection and evaluation of standard descriptors during bottom-up analysis.

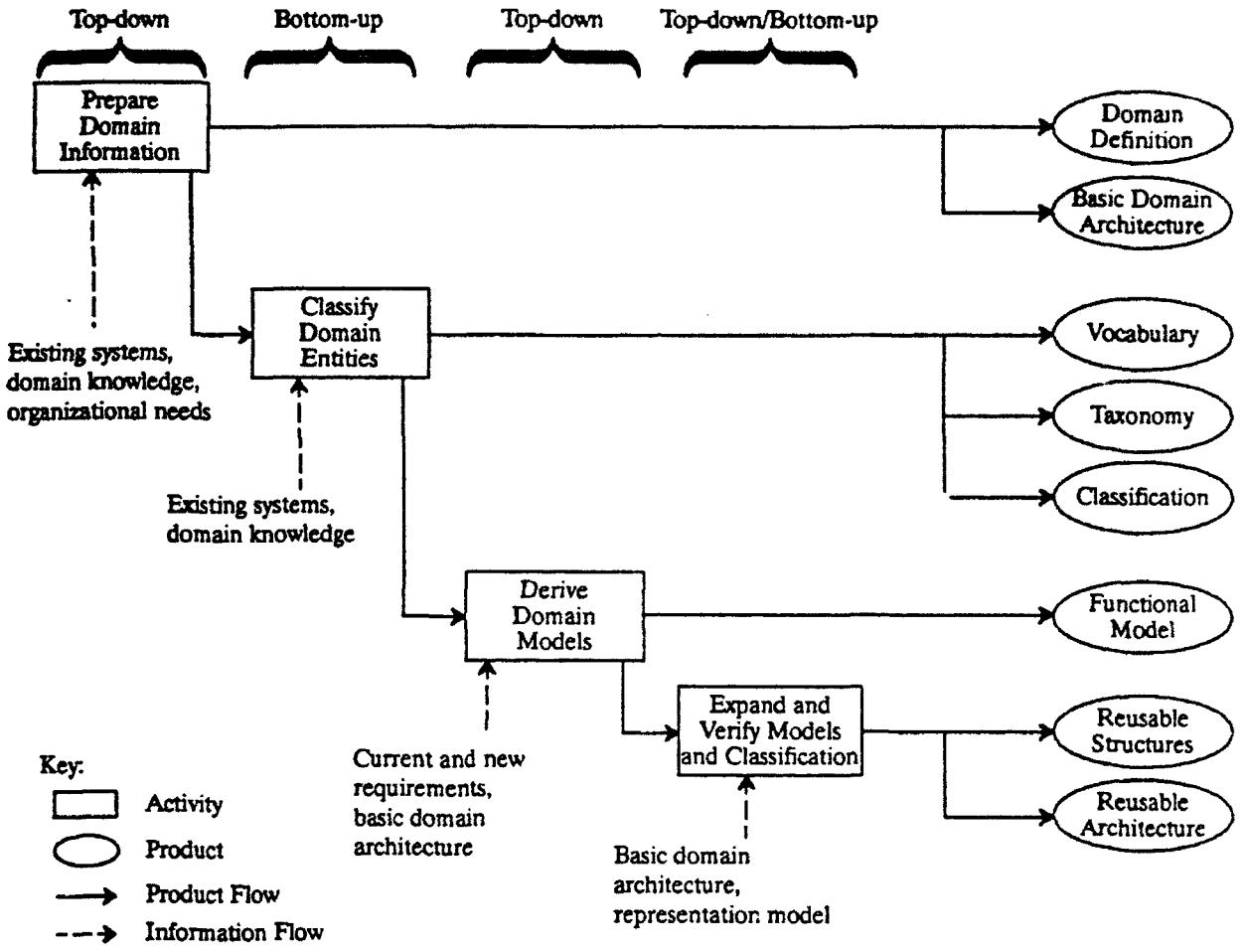


Figure 7. Prieto-Díaz's Top-Down-Bottom-Up Domain Analysis Process (1990 Version)

A bottom-up analysis activity, classifying domain entities, follows this. Here, the domain analyst examines low-level requirements, source code, and documentation from existing systems. The results are:

- A preliminary vocabulary, used as basic terminology for identification of concepts and components.
- A taxonomy.
- The classification structure. Both the taxonomy and classification structure provide the conceptual structure needed to verify and revise the basic domain architecture when deriving domain models. Application developers also use this structure when searching for reusable components.
- Standard descriptors. These form the basis for specifying standard designs and standard components. Application developers refine these descriptors to meet application needs. (They do not appear in Figure 7 because they are a means for presenting the taxonomy and classification structure rather than an independent product.)

The domain analyst uses these products in the second top-down activity, deriving domain models. This activity yields a generic functional model. This model helps a domain analyst select the proper

structural components for standardizing designs during the next domain analysis activity. The generic functional model is expressed as layers of groups of functions, integrated with the results of the bottom-up analysis. It supports design and development of new systems through composition of reusable components.

In the fourth activity, expanding and verifying models and classification, the domain analyst integrates the results of both analyses into two products:

- Reusable structures, a verified part of the functional model or classification structure seen as complete enough to be useful as a reusable component.
- A reusable architecture, which provides a framework for composing reusable structures into a legitimate software design.

The integration process consists of associating the products of the bottom-up analysis with the structures derived by the top-down analysis. Standard descriptors, for example, represent elemental components, either available or specified, by using a standard language and vocabulary. These standard descriptors define the low-level components for the reusable architecture. The result is a natural match between high-level generic models and low-level components using domain models as skeleton guides.

2.3.3 Examples

Several organizations domains are trying or have tried this methodology on an experimental basis. General Dynamics' Data Systems Division is using it in the domain of plotting empirical equations used for flight simulators. Harris' Government Communications Systems Division is using it in the domain of equipment control. Contel has used it in the command and control systems domain.

2.4 FODA

2.4.1 Overview

The Software Engineering Institute (SEI) developed the FODA approach (Kang et al. 1990). FODA is based on identifying "features" of a class of systems. A feature is a prominent, user-visible aspect of a system. Domain analysts identify both features that are common to all systems and features that distinguish individual systems or subclasses of systems within a domain.

2.4.2 Process and Products

The FODA process defines three basic activities (see Figure 8):

- Context analysis, where domain analysts interact with users and domain experts to scope the domain. The product of context analysis is a context model. Domain analysts and domain developers use it in subsequent domain engineering activity to understand domain boundaries.
- Domain modeling, which yields a domain model with multiple views:
  - A features model, which is the end user's perspective of capabilities (both common and variable) of applications in a domain.

- An entity-relationship-attribute (ERA) model, which defines the objects in the domain and their interrelationships. The model is a developer's view: domain and application developers use this information as a basis for deriving implementations of reusable components and applications.

- Data flow and finite state machine (FSM) models, which are the requirement analyst's view of the functionality of applications within a domain. The features and ERA models guide and constrain their development, that is, they reflect the commonalities and variations expressed in the features model, and the objects in the ERA model define them. Domain analysts use them subsequently in architectural modeling. Application developers also refine them into requirements for specific applications during application development.

- Architecture modeling, which produces high-level design specifications of solutions to the "problems" defined in the domain model. Architectural modeling yields a model of interacting software processes and a module structure chart. Domain developers use these products as specifications for reusable components. Application developers refine the components into products that meet their application's needs.

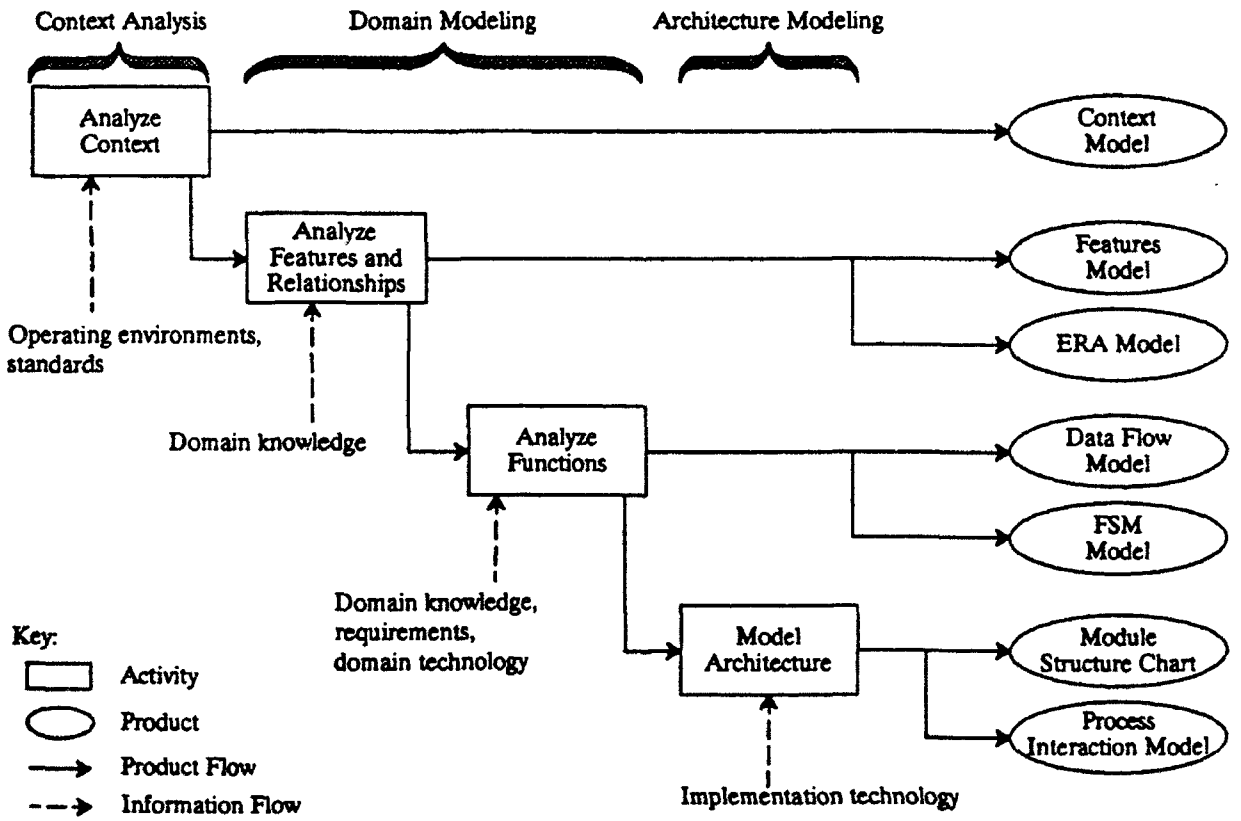


Figure 8. FODA's Domain Analysis Process

2.4.3 Examples

The FODA report illustrates the process by using the window management systems domain. The example shows in detail how domain analysts drive each of the products of domain analysis. It includes the products mentioned above.

2.5 LUBARS' SUPPORT FOR MECHANIZED REUSE USING DOMAIN ANALYSIS

2.5.1 Overview

Lubars' domain analysis approach derives from his previous work on mechanizing reuse using design schemas in IDeA (Intelligent Design Aid), a reuse-based design environment to assist software designers (Lubars and Harandi 1987; Lubars 1987). It supports reuse of abstract software designs. Lubars created IDeA and its successor, ROSE-1, as a proof-of-concept tools to demonstrate the reuse of high-level software work products other than source code. Lubars' research has focused on representation models for software designs, the objective being software design reuse. IDeA provides mechanisms that help users select and adapt design abstractions to solve their software problems. Before IDeA can provide support for design reuse, however, a designer experienced in certain application domains must populate IDeA's design reuse library with the schemas for these new domains. This manual process is extremely difficult and tedious. To reduce the effort required to identify, select, and characterize designs for the IDeA library, Lubars developed a domain analysis methodology.

2.5.2 Process and Products

Lubars' approach to domain analysis resembles Prieto-Díaz's early model (Prieto-Díaz 1987). It goes through a similar process of identification, selection, abstraction, and classification of objects, operations, and other domain items. A domain engineering activity (Lubar's term for what the other approaches discussed call domain implementation) follows domain analysis. Figure 9 shows the process. Unlike the other approaches, it has no preliminary analysis phase.

Lubars' process has three stages. Each stage ultimately results in identifying common abstractions pertinent to that stage:

1. ***Analysis of Similar Problem Solutions.*** This yields characterizations of solutions for particular classes of problems in the application domain.

2. ***Analysis of Solutions in an Application Domain.*** This groups the characterizations from stage 1 to produce characterizations of particular application domains.

3. ***Analysis of an Abstract Application Domain.*** This generalizes the characterizations from stage 2 to represent classes of related application domains.

Lubars' heuristics aim at characterizing generic solutions to common problems in a domain, and providing a reasonable mapping between problems and solutions to make reuse practical. Lubars uses design schemas as mechanisms for mapping problems to solutions. He goes farther than the other approaches in trying to identify commonalities or similarities in domains other than the one of interest. Lubars addresses the issue of reuse across domains. His approach covers both "vertical" reuse (involving related components of a subsystem) and "horizontal" reuse (identification of components

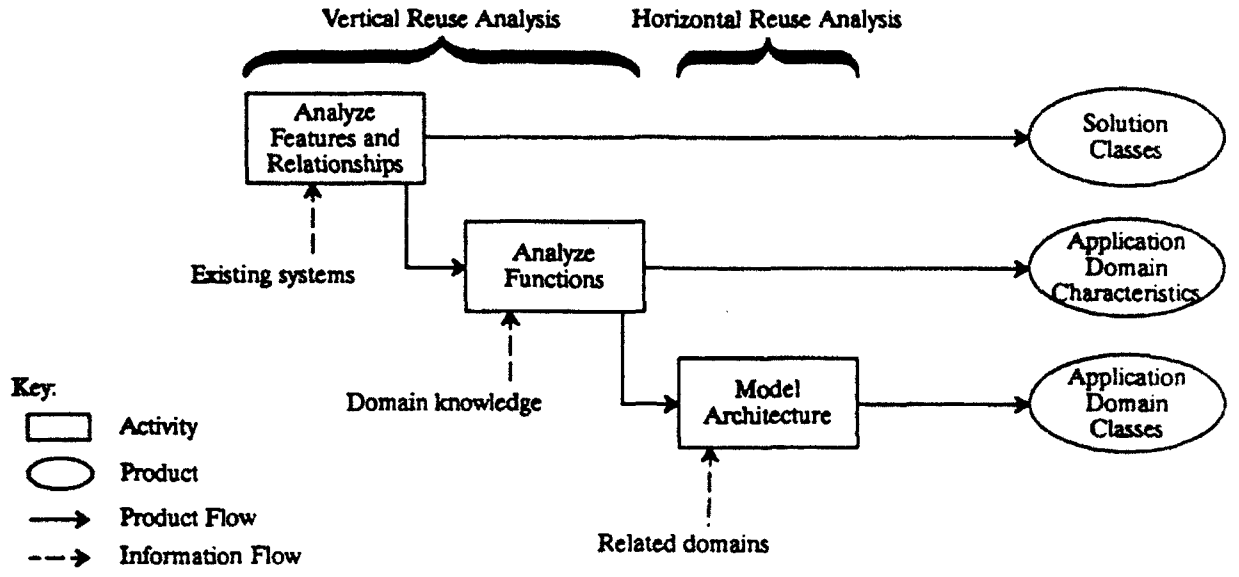


Figure 9. Lubars' Domain Analysis Process

for a particular module). Analysis stages 1 and 2 in his heuristics identify vertical components. Stage 3 aims at finding horizontal components. Lubars introduced the clear separation between each of the three stages because he believed it made his methods amenable to modeling and possible automation.

Lubars' domain analysis approach concentrates on commonalities. Domain analysts only identify variations informally. The activities that formally define them are part of domain engineering. There, domain developers assemble the abstractions into a type hierarchy, identifying discriminators among the types. These discriminators form property trees. A mapping between the data types and the properties formally defines the domain variations.

2.5.3 Examples

Lubars has successfully characterized common designs in the domain of tax computations, mainly to illustrate his method and to demonstrate its application. He has characterized and encoded a set of designs into IDeA and ROSE-1 to demonstrate reuse at the design level.

2.6 KAPTUR

2.6.1 Overview

KAPTUR is a domain analysis support environment. The need to understand and maintain large software systems with long life spans motivated its development. It is the product of a project for NASA/Goddard Space Flight Center to investigate reusability of their mission control software. They use mission control software in long satellite missions, and it typically persists through new technologies and discoveries. NASA updates it continuously and demands high modularity.

A model of competing/cooperating interests between demand and supply of reusable software (Bailin and Moore 1989; Moore and Bailin 1991) is the basis for KAPTUR concepts. Demand for reusable components provides domain developers with requirements for developing software intended to be

reusable. The availability of reusable software, in turn, stimulates systems developers to create new systems out of reusable components. KAPTUR supports this view by capturing domain information from existing systems and new requirements and making it available to domain developers. This approach aims at integrating domain analysis into a reuse-based software development process.

KAPTUR's goal is to support reuse of "software assets" (a term for a reusable artifact) by capturing design decisions and rationales that went into their development. KAPTUR's knowledge base stores reusable assets with their corresponding rationales, underlying issues, and lists of differences and similarities to other assets. The creation of this knowledge base is a time-consuming and involved task. It is the product of domain analysis. Bailin proposes an evolutionary and incremental approach to building the knowledge base. He sees domain analysis as complementary and parallel to systems development, that is, domain knowledge is acquired as systems are developed. He believes that the supply and demand model facilitates this view.

2.6.2 Process and Products

KAPTUR supports reuse through generic architectures. Domain analysts use ERA, data flow, object communication, state transition, and stimulus-response models to specify architectures. Each captures a different architectural view and provides for expressing possible variations. Once an application developer has specified how his application varies with respect to the models, KAPTUR partly automates the refinement of a generic model into an application.

Bailin proposes an overall domain analysis process for families or classes of systems and a more narrow approach to identify commonality and reusability of components organizations integrate them in an ongoing software development process. He intends the former for initial population of KAPTUR's knowledge base and the latter for expanding and evolving it. Figure 10 shows the overall process.

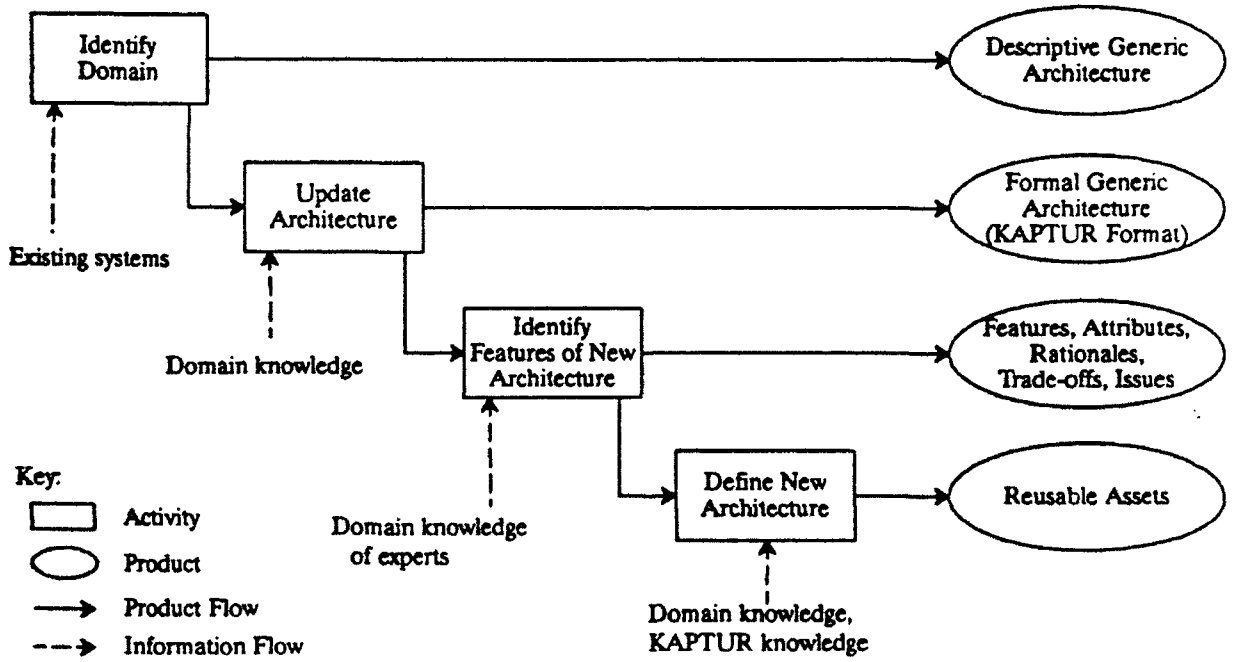


Figure 10. KAPTUR Domain Analysis Process

In the overall domain analysis process, the first step is to examine existing systems descriptions in a domain to identify generic architectures. The domain analyst then formalizes this information using the models of KAPTUR; the formalization facilitates subsequent activities. He extracts features, attributes, rationales, trade-offs, and issues from the generic architectures based on their differences (and similarities). A review with domain experts follows the process. This extraction and comparison process is an important feature of this approach because it helps to derive a criterion for classification. The next step in the process is to identify reusable assets from the architectures, their probable source, and their documentation, if available. Domain analysts review documented assets and architectures with domain experts and then load them into the knowledge base. When new components are identified or a new system is analyzed, KAPTUR supports expansion of the knowledge base through a similar process.

2.6.3 Examples

NASA/Goddard Space Center is using KAPTUR, currently in its second release, on an experimental basis to analyze mission control software. Plans are underway for integrating KAPTUR into an operational software development environment.

2.7 PROCEDURAL COMMONALITY

Table 2 lists the domain analysis processes for each of the approaches analyzed. The table summarizes the processes illustrated in Figures 2, 3, 7, 8, 9, and 10. It lists the four basic processes for each of the six domain analysis approaches. On the top row are names of generic processes that capture the essence of the activities in each column. The table abstracts essential features from common procedures. The result is an abstract view of the basic activities in domain analysis. The following four activities characterize the domain analysis process:

1. Study the domain.
2. Analyze domain entities.
3. Compile, abstract, and structure domain knowledge.
4. Generate reusable structures.

The activities in the first column can be characterized as "identify," "describe," "define," "scope," "provide context for," and "select information sources for" the domain. The Consortium selected the term "study domain" to characterize the nature of these activities. The overall objective is to observe and study the domain and to characterize its nature informally.

The activities in the second column can be summarized by the term "analyze domain entities," where analyze stands for "identify, bound, qualify, and classify." Domain entities include objects, operations, relationships, features, problems, and characteristics.

The third column's activities are basically creative. They deal with the concept of creating a knowledge base, a domain model, product requirements, functional components, solutions, and new features. The phrase "compile, abstract, and structure domain knowledge" is representative of these activities.

The last column can be identified with the phrase "generate reusable structures." The activities produce canonical/generic requirements, designs, architectures, models, structures, abstractions, and solutions.

The six domain analysis approaches contain specific instances of these activities. This bottom-up procedural analysis of the approaches provides some insights on a high-level model for doing domain analysis.

Table 2. Common Procedures for Six Domain Analysis Approaches

Approach	Corresponding Common Activity			
	Study the Domain	**Analyze Domain Entities**	**Compile, Abstract, and Structure Domain Knowledge**	**Generate Reusable Structures**
Jaworski	Describe domain	Qualify domain	Create knowledge base	Develop canonical requirements
Synthesis	Define domain	Define decision model; define product requirements	Define product and process requirements; define decision model	Define product requirements; design product
Prieto-Díaz	Prepare domain information	Classify domain entities	Derive domain models	Expand and verify models and classification
FODA	Analyze context	Analyze features and relationships	Analyze functions	Model architecture
Lubars	Not applicable	Analyze similar problems	Analyze domain solutions	Analyze abstract application domains
KAPTUR	Identify domain	Update architecture	Identify new features	Define new architecture

3. SIMILARITIES AMONG APPROACHES

This report is primarily about contrasts among domain analysis approaches. Understanding these contrasts requires some knowledge of characteristics that the approaches share. Researchers in domain analysis have created quite different means to achieve an end, but they share many opinions on what that end should be. This section discusses what the approaches have in common.

3.1 THE DEFINITION OF "DOMAIN"

A most basic question to address is whether all approaches agree on what the term "domain" means. In fact, they do not (see Section 4.1), but their definitions do possess some similarities. Prieto-Díaz defines a domain as follows (Prieto-Díaz 1990):

> In the context of software engineering it is most often understood as an application area, a field for which software systems are developed. Examples include airline reservation systems, payroll systems, communication and control systems, spread sheets (sic), [and] numerical control.

The implicit assumption is that more than one system exists for the application area. These systems might be the result of wholly different projects that have independently built similar systems for different customers. They might also be versions of a single system, evolved over time based on changing customer needs. The origins of the systems are unimportant; what matters is variations in observable behavior.

Each system performs similar types of operations on similar types of objects. This is true because of the nature of the area. It can safely be said that any airline reservation system will deal with airplanes, and that any payroll system will deal with money. Moreover, each system operates under certain constraints because of the laws of the area. A communication system can expect certain types of messages to be propagated at the speed of light. Calculations in a spreadsheet system behave in accordance with the rules of mathematics.

Despite these similarities, applications within an area vary. Variations may occur because of differences in offered functions, differences in implementation strategies, the nuances of various types of hardware on which they are implemented, or for many other reasons. However, a certain central abstraction (or perhaps a set of abstractions) always characterize a domain. This central abstraction gives an intuitive feeling for what the domain is and the types of applications that can exist within it. All approaches to domain analysis share this characterization of a domain. (As Section 4 will show, they differ about whether the similarities or the differences are more important.)

This central abstraction can be viewed as a "problem-level," as opposed to a "solution-level," conceptualization. This defines another common view of domains. All approaches see a domain as addressing a set of problems. Applications in the domain provide solutions to those problems. Furthermore, people need to describe problems in the domain independent of any solutions to them. In a C^3I domain, for instance, they should be able to talk about communication protocols without

- ***The Capture and Formalization of Domain Knowledge.*** Software systems are not necessarily in well-understood domains. To limit domain analysis to such domains would reduce its utility. Therefore, the approaches all provide ways to capture information about a domain in such a way that it is sufficiently formal to satisfy whatever needs are expected of it. This does not always result in the "optimal" domain, but rather in one that is sufficient to leverage productivity, and also evolvable as new knowledge is gained about the domain.

3.4 SHARED CONCERNS

A shared concern is something seen as important to overcome to make domain analysis more effective. All approaches share the following concerns:

- ***The Need for a Precise Definition of the Context, Products, and Process of Domain Analysis.*** The benefits of domain analysis can be best achieved—or achieved at all—if people have rigorous specifications of what they must do and how they must do it. This does not imply that the approaches agree on what the context, products, and process of domain analysis should be.

- ***The Transfer of Domain Analysis Costs to Customers.*** Performing domain analysis introduces costs that are not normally chargeable to customers, even if they may benefit (building and maintaining a reuse library, for example, or simply taking the time to do the analysis). It is generally assumed that a single organization will be responsible for implementing a domain. That organization should not transfer the entire cost to a single customer (i.e., the one who requests the first application). The potential competitive advantage gained from domain analysis lies in reuse of software across a set of applications (or revisions), so an organization should apportion the costs of domain engineering across the expected number of customers for the domain. Anyone planning a factory faces an analogous situation, but software engineering economics is a much less mature field than for other engineering disciplines.

- ***The Validation of Domain Analysis Results.*** Domain analysis yields an abstract model corresponding to combinations of hardware, software, and humans. All approaches agree that there is a need to establish confidence that the model accurately reflects the domain. None has yet found an entirely satisfactory way of doing so.

- ***The Reduction of the Up-front Costs of Domain Development.*** A customer who contracts for software will not be anxious to have the developer spend extra time and money to develop an entire reuse library for the domain in question. (Whether the contractor **should** be accepting of this is quite another matter, but outside the scope of this report.) The lower the up-front costs of developing a domain, the more palatable a customer will find the situation.

This page intentionally left blank.

4. COMPARISON CRITERIA

This section presents the criteria by which the domain analysis approaches surveyed are distinguished. Each criterion represents a decision about how an organization should approach domain analysis.* Associated with a criterion are a set of possible resolutions of that decision. These resolutions result from analysis of existing systems. As such, they represent ways that researchers have seen fit (so far) to undertake domain analysis.

These are not the only criteria distinguishing domain analysis approaches. They reflect differences among the methods presented in this report. Including another method might have required additional criteria. However, this report includes only those criteria that relate to the contextual factors in Section 1.3. (The discussion for each criterion describes that relationship. Table 3 summarizes it.) A criterion such as the products of a given domain analysis approach is uninteresting. It does not uncover the rationale behind why researchers felt the various products necessary. Knowledge of such products supports the discussion but is not the central point of the analysis.

Table 3. Relation of Criteria and Contextual Factors

Criterion	Contextual Factor				
	Software Process Needs	**Existing Software Base**	**Business Objectives**	**State of Domain Knowledge**	**Intended Use of Information Repositories**
Definition of "domain"			✓		
Determination of problems in the domain		✓	✓	✓	
Permanence of domain analysis results			✓	✓	
Relation to the software development process	✓				
Focus of analysis		✓		✓	
Paradigm of problem space models					✓
Purpose and nature of domain models		✓		✓	✓
Organizational models of domains and projects	✓		✓		✓
Approach to reuse	✓		✓		
Primary product of domain development	✓				✓

* "Organization" can mean any group, from an individual developer to an entire company, undertaking or considering domain analysis. All meanings are relevant across the approaches.

Some researchers consider domain analysis part of a larger process called domain engineering. In this view, domain analysis yields a specification. A separate subprocess of domain engineering (often called domain implementation) transforms this specification into products that are directly useful in application development. To simplify comparisons, this section concentrates on domain analysis. Except where specifically mentioned, it does not consider domain implementation.

Table 4 summarizes the criteria. The remainder of this section discusses them in detail.

Table 4. Summary of Comparison Criteria

Criterion	Meaning	Choices
Definition of "domain"	What a domain encompasses, how that influences what is considered a domain, and how organizations satisfy business goals accordingly.	• Application area • Business area
Determination of problems in the domain	The approach used to arrive at the set of problems that make up a domain.	• Problem-oriented • Solution-oriented • Problem/solution-oriented
Permanence of domain analysis results	Whether products of domain analysis evolve.	• Permanent • Mutable
Relation to the software development process	How domain analysis activities fit into a software process model activities (or vice versa).	• Pre-requirements, dependent • Pre-requirements, independent • Meta-process
Focus of analysis	The fundamental concept on which analysts focus during analysis.	• Objects and operations • Decisions
Paradigm of problem space models	The fundamental concept emphasized by the problem space model the analysts derive.	• Generic requirements • Decision model • Both
Purpose and nature of domain models	Intended uses of the products of domain analysis.	• Repository • Software specification • Process specification
Organizational models of domains and projects	Possible organizations a company might use to maximize the potential of domain analysis.	• Circumstance-driven • Project-driven • Domain-driven
Approach to reuse	Strategies for exploiting the reusable components generated during domain analysis and implementation.	• Opportunistic • Systematic
Primary product of domain development	Most significant product resulting from domain implementation, guiding how other products will be used.	• Reuse library • Application engineering process

4.1 THE DEFINITION OF "DOMAIN"

Perhaps the most basic difference is the lack of general agreement on what a domain is. Section 3.1 noted certain similarities among approaches. A domain is a set of related problems. In a mature domain, solutions to those problems exist, i.e., applications in the domain. This problem/solution space dichotomy is common to all domain analysis approaches. However, it still begs the question of what forces drive the problems and solutions. Two definitions of "domain" used in domain analysis approaches reflect this:

1. ***Application Area.*** A domain can be seen as an application area. In this view, the subject matter of the domain is of paramount importance. People tend to equate the applications with the domain—the domain of stack packages, for instance. Notions of problems in a domain come from problems the applications solve.

2. ***Business Area.*** A domain can also be seen as a business area. Such a domain not only contains applications, but the external forces that motivate the domain constrain it. These include:

   - ***Systems Engineering Concerns.*** In other words, domain analysis can be considered in the context of a larger process.

   - ***Economic Factors.*** These include the cost/benefit considerations of implementing the domain. They relate to the business objectives of each organization within a company and how the domain fits into that focus.

An organization ultimately performs domain analysis for "business reasons"—increases in productivity, decreases in error rate, etc. It therefore cannot entirely separate business objectives from domain analysis concerns. Still, the approaches appear to have three points of contention. The first is the degree to which business concerns drive the analysis process. The approaches that make a domain a business area advocate using customers' needs, both real and anticipated. The approaches that make a domain an application area tend to focus more on developer perspectives—that is, more emphasis on function than on rationale. (All approaches realize that customer inputs are needed. However, some focus more around developers' concerns. See Section 4.5.) Although customers' needs ultimately mandate the course to follow, such needs are not easily quantified; analyzing existing applications is a less abstract task.

The second point of contention is what an organization would consider a domain. In the application area view, any set of related programs is a domain. Businesses, however, want to focus on larger, more profitable systems. Also, it is usually easier to identify a business need with a complete system than with some software package. This is not universally true, as companies that market mathematical and graphical software can attest. The implication, though, is that a business-oriented domain is more likely to possess these characteristics. In the business area view, then, a stack family is not a domain.

The third point is what to consider a subdomain. Domain analysis approaches with an application-oriented view often describe lattices of domains. Each level uses applications in domains from the levels immediately beneath it. A domain therefore has an architecture composed of a set of interrelated subdomains. This type of architecture can be used in the business-oriented view, but the approaches that adopt this view define subdomains differently. In these approaches, a subdomain is a subset of the business area that comprises the entire domain. This philosophy permits gradual, planned growth of a business area from a small but economically viable subset (perhaps satisfying only a single customer's needs) to the full-blown capability.

4.2 THE DETERMINATION OF PROBLEMS IN THE DOMAIN

All the approaches contain a view of a domain as a set of problems with corresponding solutions for each problem. It is possible to arrive at this view in three ways:

1. ***Problem-oriented.*** Domain analysts can first define a set of problems, for which they then derive solutions (applications or specifications thereof). The modeling constructs used in the initial stages of domain analysis concentrate on problem-level concepts. The analysts subsequently refine them into concepts appropriate to specifying the solution (usually software design and implementation constructs).

   An initial informal definition of domain scope, subsequently refined into a more formal model, typically characterizes such approaches. The model is a specification of problems; it determines what solutions are possible and viable.

2. ***Solution-oriented.*** Domain analysts can start by examining applications, from which they determine the common problems. This may sound backward, but it has much justification in the realities of domain analysis. Some researchers assume that domain analysis is useful only after a certain number of applications have been built—specifically, enough to give sufficient knowledge to make construction of a domain worthwhile. In this view, domain analysis happens after enough applications in a domain have been built to give people a firm understanding of the principles underlying the domain.

3. ***Problem-/Solution-oriented.*** The approach uses problem-oriented analysis for some products and solution-oriented analysis for others, based on an assessment of which seems most appropriate. Problem- and solution-oriented analysis thus occur concurrently.

The relative merits of each approach depend on an organization's domain analysis objectives and the existing software base available. Problem-oriented determination concentrates initially on domain concepts rather than on software development concepts and domain implementation details. This seems likely to yield domain models that are consistent with specific business needs and to derive problem statements that are consistent with the current and future external requirements of a domain.

Solution-oriented determination seems advantageous for the reuse and reengineering of existing components. Consider a project tasked to create a reuse library from a given a set of components. Domain analysis under the second choice proceeds by quickly creating an index into the components (e.g., faceted classification). The project can use the index to catalogue the components in a reuse library. Intuitively at least, the process seems faster and less error-prone than the first choice—specifying the components' domain and refining that specification toward the components themselves. On the other hand, it is hard to infer the business needs of a domain by studying applications. Such information therefore must come later in the domain analysis process.

The first choice demands that the state of domain knowledge must be such that standard problems are already recognized, or can be formulated, without examining applications. This implies greater domain maturity than the second choice. Both approaches end up with the same results, however.

No approach is pure. Domain analysts must draw on their knowledge of both existing applications and domain concepts. Problem-/solution-oriented determination takes advantage of this by using problem-oriented derivation techniques for those products best defined by studying domain concepts, and solution-oriented derivation techniques for those best defined by studying existing applications.

4.3 THE PERMANENCE OF DOMAIN ANALYSIS RESULTS

Each approach has an associated process that defines when domain analysts first develop a product, whether it can evolve, and if so, how. The approaches considered in this report define two possibilities:

1. ***Permanent.*** The process has no provision for evolving its products. The approach assumes that the results will be complete and correct before the first use. The presumption is that any domain mature enough to be amenable to domain analysis is stable. The problems it poses will not evolve, nor will its terminology, laws, and the like. In short, it needs no evolution. This does not preclude evolution of applications in the domain. Technological advances and economic concerns may lead to new solutions for problems in a domain. These solutions may change over time as long as they continue to satisfy the specification of problems. The specification, however, is expected to be correct initially.

2. ***Mutable.*** The process has steps that allow domain analysis products to evolve over time. Domain knowledge gained both from use of products in the domain and from external influences (i.e., new technology) is the basis for evolution.

An organization must believe a domain to be fully mature before committing to an approach endorsing the first choice. Such domains exist, usually in long-established branches of the physical sciences—equations governing thermodynamics or the laws of perspective, for instance. These domains are application areas rather than business areas. They are narrow in scope, and a company cannot base its business on them. If the domain proved profitable, competitors would jump into the market, and the company could not maintain its competitive edge. A mutable domain is a more viable business area. A domain analysis approach that supports mutability will help an organization plan the evolution of a domain based on business objectives.

A "permanent" domain is still useful as a subdomain, in the sense of an application area view of a domain. It provides organizations with access to parts that can simplify other implementations (the Common Ada Missile Package, CAMP [CAMP 1987], illustrates this point). Moreover, even if problems in the domain are universally recognized, organizations will probably incorporate improved solutions from time to time.

4.4 THE RELATION TO THE SOFTWARE DEVELOPMENT PROCESS

The products of domain analysis are not an end in and of themselves. Organizations will use them as part of some other process. What that process can be, and where domain analysis fits into it, are also issues to consider. The possibilities identified are as follows:

1. ***Pre-requirements, Dependent.*** An approach can make domain analysis a pre-requirements activity of a specific software process model or software development method. The approach integrates domain analysis activities into those of a model; the outputs of each domain analysis activity are intended as inputs for specific activities of the software process or method in use. Domain analysis occurs during or following the systems requirements phase but prior to the software requirements phase (hence, "pre-requirements"). Domain implementation can precede, or be part of, subsequent application software design activities. The resulting reusable parts are then available during the software requirements and design phases of any other projects writing software for the domain.

2. ***Pre-requirements, Independent.*** Domain analysis can be a pre-requirements activity, but be independent of the life-cycle model. The products of domain analysis are intended to be general enough to be used in a variety of processes. However, domain analysis is still to occur prior to software requirements analysis (so the model must have a requirements phase).

3. ***Meta-process.*** Like software, an organization can design, implement, and evolve a process model. A process is termed a "meta-process" if its goals include process construction. Domain analysis can be part of a meta-process. To be so, it must fulfill the following conditions:

   - The process for domain analysis and implementation must be separate from the process for application development. It must be possible to do domain engineering without doing application development, and vice versa.

   - The results of domain analysis and implementation must influence the process for application development. "Influence" might mean mechanizing or reordering activities.

The only real constraint is that domain analysis (and, perhaps, domain implementation) take place prior to the activity that uses its products. Software process needs dictate that products required as inputs to an activity be produced prior to the onset of that activity. Domain analysis approaches that emphasize evolution of a domain relax (though not fully) even this constraint. A preliminary version of a product may be enough to understand if it will, in a subsequent activity, fulfill some role.

The more interesting issue, directly related to an organization's software process needs, is whether a domain analysis approach can fit that organization's software process model. The first choice's viability is linked to an organization's chosen model. The second and third choices are intended to be useful with arbitrary models, although the approaches they represent have not necessarily been verified across a range of models.

However, recent research on software processes has discussed the need for constantly maturing, self-improving processes that incorporate feedback on process quality. This is exemplified by the SEI's work (Humphrey 1989). It specifies five levels of software process. The lowest level, termed "chaotic," means an organization uses no well-defined procedures. The highest level, termed "optimizing," means an organization's software process is well-defined, repeatable, manageable, and self-improving. That is, it has associated metrics to facilitate identifying and correcting problem areas.

Many researchers believe domain analysis can help an organization reach level 5. The formalization of domain information can describe how to measure the efficacy of activities, for instance, leading to identification of trouble spots within a process. An organization that seeks both to mature its process along the SEI scale and to use domain analysis must consider how a domain analysis approach will contribute to, or hinder, process maturation.

A pre-requirements, process-dependent approach can be part of a level 5 process. An organization can presumably use pre-requirements, process-independent approach in conjunction with arbitrary processes. A meta-process approach can ensure that domain analysis produces an appropriate process; that is, domain analysis can lead to a level 5 process for developing applications in a given domain. Note that the process for domain analysis might not be level 5, even though the process that **results from** domain analysis is. An organization must still define a suitable process for domain analysis and domain implementation. However, given that it has a level 5 process for building applications (i.e., for fulfilling customer contracts), it is closer to achieving its goal than the processes of the other two types.

Not everyone considers the SEI's scale a good measure of process maturity (see [Bollinger and McGowan 1991]). In this case, the dominant software process consideration is whether a domain analysis approach delivers products that support software process needs.

4.5 THE FOCUS OF ANALYSIS

Domain analysis strives to uncover certain fundamental views of a domain. Each approach identifies multiple types of views. Among these is a "focus of analysis." This is the view that, out of all types identified, is central to the analysis approach. It determines how domain analysts will undertake their tasks, and it shapes the products of the analysis activities. The different focuses are as follows:

1. ***Objects and Operations.*** The analysis can center around objects and operations among similar systems. Depending on the approach, this can mean focusing on requirements, architectural design, or detailed design. In requirements, the analyst concentrates on identifying objects in the domain (devices, users, etc.) and stable functional and nonfunctional requirements for those objects. The analyst focusing on architectural design looks for process models and design structures that characterize all applications in the domain. The analyst focusing on detailed design uncovers module interfaces and the operations of those interfaces.

   No matter what level the focus, the domain analyst concentrates on what is common in the domain. All applications share the objects and operations. Analysis therefore focuses on similarities.

2. ***Decisions.*** The analysis can concentrate on decisions that application developers need to make to derive an acceptable solution to a problem in a domain. The domain analyst uses domain concepts to define a means to differentiate problems in terms of decisions that lead to solutions. He therefore concentrates on how applications differ. This is a focus at the domain requirements level only. The design and implementation considerations of the decisions are a secondary focus of analysis.

In all approaches, decisions ultimately become a focus, although not always **the** focus. Describing a domain without identifying differences limits the applicability of a domain implementation. In approaches where domain analysts study similarities first, they do not fully formalize them. This suggests that people believe similarities can be understood at a high level of abstraction.

Analyzing objects and operations among similar systems is most useful when studying properties of existing systems. In particular, analyzing objects and operations at the detailed design level requires greater accessibility to a large software base containing such objects than analyzing requirements or architectural design. For instance, the domain analyst can consult domain experts to determine common application properties. However, concentrating on similarities of existing applications does not account for anticipated changes in the domain.

Analyzing decisions is more useful than analyzing objects and operations when considering new customer requirements for a domain as well as its current properties. The domain analyst can examine existing systems and enumerate their differences in the model prescribed by the approach. He expresses the differences using domain-level concepts. Adding new customer requirements therefore requires extending the domain model to include new concepts. Since the domain model is a statement of problems, such extensions depend less on existing applications than on insights into the problems.

4.6 THE PARADIGM OF PROBLEM SPACE MODELS

Correlated to the focus of analysis is what the problem space model emphasizes (that is, the products of domain analysis related to problems in the domain) derived during domain analysis. Although the analyst brings a particular view to bear during the analysis phase, that view is not necessarily the overriding emphasis of the resulting products. The choices are as follows:

1. ***Generic Requirements.*** The problem space model can emphasize generic, reusable requirements. This is an emphasis on commonalities. The model shows a view of the problem space primarily in terms of what is similar among all systems.

2. ***Decision Model.*** The problem space model can be a decision model. Its most important component shows how to distinguish applications based on a set of decisions, each of which resolves some facet of a problem in the domain.

3. ***Both.*** The problem space model can also consider both but emphasize neither. Both views are important but at different times and for different people.

The relative advantages and disadvantages of the problem space model emphasis depend on who will use the products and how. Customers need to understand differences among systems in a domain, so that they may define the requirements for the one they desire. Developers need precise characterizations of a domain's unvarying aspects to guide them in selecting architectures, algorithms, and data structures.

The paradigm of a problem space model must be consistent with the intended use of that model, especially with respect to the approach for reuse. Some approaches assume that reuse stems from a realization that a particular subproblem of an application has a solution already extant. The implementor first recognizes the problem as matching (loosely) one previously solved, then tailors the matched solution to the needs of the problem at hand. This makes the decision model of secondary importance to the generic requirements.

An opposing view assumes that reuse can be systematized based on the domain model (see Section 4.7). The domain analyst merges existing applications with a process that shows how to extract an application that corresponds to a particular set of decisions. The primary consideration here is how to distinguish among those applications, both during application generation and during the merging. This view therefore emphasizes the decision model.

Equal consideration of both reflects a philosophy of refining systems from canonical designs. When no complete implementation products exist, developers require specific guidelines to ensure that they implement applications in ways appropriate to the domain.

4.7 THE PURPOSE AND NATURE OF DOMAIN MODELS

The approaches use the term "domain model" to refer to those products that result from domain analysis. The previous section discussed that portion of the domain model related to the problem space. However, the complete model also has a focus of its own that varies between approaches:

1. ***Repository.*** The domain model can serve as a repository of domain knowledge. It supports queries about the domain: what applications are in it, what physical laws constrain it, what relationships exist between objects and operators, etc.

2. ***Software Specification.*** The domain model can be a specification for software products. It guides developers in building reusable components.

3. ***Process Specification.*** The domain model can be a specification for a software development process and environment. This is a generalization of the previous item. The reason is that a process, to be useful, must specify both the products it requires as inputs and what products it produces as outputs. The latter is exactly what a software specification describes. The process also may define how to effectively use those products. Furthermore, the domain analyst can study the process to determine the cost-effectiveness of automating portions of the process. This serves as a specification for an environment.

Experimental implementations of repositories (e.g., LaSSIE, a knowledge-based information retrieval system, [Devanbu et al. 1991] and KAPTUR) provide an intelligent assistant for domain analysts, domain developers, and application developers in an automated domain analysis environment. Their creators acknowledge a potential scale-up problem they have not yet faced.

A software product specification and a software development process and environment specification differ in terms of an organization's intended use of information repositories. If the goal is to have a library of parts that an application project can search, then a specification of software products suffices. If the goal is to also provide mechanisms for adapting and composing those products rather than leaving the adaptations to the skills of developers, then the organization must also have a process specification describing how these adaptations are to occur.

Specifying a software development process and environment has the highest potential payoff. The resulting standardization lessens the cost of application development more than the other two choices. However, an organization must be willing to invest in making development an application-domain-oriented activity rather than a software design activity. The investment involves defining the process and (possibly) automating the environment.

However, specifying only software products seems preferable in two circumstances. First, it may be quite reasonable for an organization that is new to software reuse. Specifying a process and environment assumes a willingness to adopt a mature reuse capability, an advance that may be too radical for some organizations.

Second, having the domain model specify only software products may be preferable in immature domains. These have several characteristics that lessen the viability of a process. First, the domain analyst may not be able to predict the range of variations accurately. Second, one important component of a process is the ability to validate and assess products; techniques for doing so are less likely to exist in immature domains. Third, the usefulness of a process depends in part on the ability to capture engineering judgment about the domain. In immature domains, where such judgment is not well formalized, the process may require large amounts of human insight (in the form of software design and implementation decisions), making its value little more than that of the products alone. Until the domain matures (aided, one hopes, by domain analysis) the process can say only so much. An organization risks low return on investment when trying to formalize a process in an immature domain.

4.8 THE ORGANIZATIONAL MODEL OF DOMAINS AND PROJECTS

Although domain analysis can be valuable for a single project, researchers believe its real payoff will come from organization-wide use of its products.* Such analysis and implementation of domains requires a significant corporate commitment. Once an organization makes the decision to begin analysis, it makes devote resources in the form of domain experts, end users, software developers, etc. (not to mention hardware resources), to the cause. Nor, most researchers agree, can the effort end after an initial implementation of the domain. Mechanisms must be in place to gather and incorporate feedback, so the domain may evolve in response to enlarged visions and ever-changing needs.

This raises the question of how a corporation should arrange its organizations to facilitate effective domain analysis. The following are possibilities:

1. ***Circumstance-driven.*** Domain analysis can be a subproject of whatever project first recognizes the potential benefits from the systematization of a domain. The project would analyze and implement the domain, then make it available to other projects needing it.

2. ***Project-driven.*** Application projects can be customers of independent domain organizations. For each domain of interest, a company would establish a separate, centralized organization. This organization would have full responsibility for models of, and standard parts for, the domain. Projects that need to build applications within the domain would use these models and products and provide feedback.

3. ***Domain-driven.*** Projects can be a component of a domain organization. In this model, a company would initiate an application project in a specific organization within a company based on the relevance of the project to the domain controlled by that organization. Unlike the project-driven organization, the organization that is responsible for the domain manages application projects.

The importance of this issue lies in the need for projects to access domains, both to obtain reusable components and to help evolve the domain. Whether or not the technical problems of domain analysis are ever solved, the technique is likely to have little impact unless it is made bureaucratically effective. Corporate management must actively encourage access to and evolution of the domain. This is outside the scope of what a single project whose original purpose is something other than domain analysis can accomplish. Domain analysis under a circumstance-driven organization is therefore likely to be of low value beyond the scope of the project that performs it. It may still be worthwhile, especially on large, long-lasting projects, but spreading its benefits to other projects may prove cumbersome.

A domain-driven organization can put communication channels in place to foster and provide feedback on the efficacy of domain models and implementations. A domain-driven organization can tailor its structure so that its component projects have easy access to those responsible for maintaining the domain (or, indeed, can have its component projects accept some of this responsibility). This is much harder to achieve in a project-driven organization where one project is responsible for maintaining a domain but has no authority over, or direct responsibility to, those projects that use the domain. The creation of the UNIX operating system provides anecdotal evidence for the viability of a domain-driven organization. Its creators believe that much of UNIX's success stems from its use for several years within a single organization before being released to a larger outside population (Ritchie and Thompson 1974). Its creators evolved it based on the needs of its customers—people within a small group at AT&T—from whom they could expect feedback.

* The Consortium is not aware of any data supporting the beliefs in this paragraph, but they seem to be universally held. See (Prieto-Díaz and Arango 1991).

An organization's business objectives help it choose a model. The organization must believe that the model it chooses will be cost-effective. Several organizations might need to reorganize so that projects are under the proper control, or so an autonomous organization can control a domain. A company must consider its customers in making these decisions. For example, suppose it uses the domain-driven approach and has several organizations that each support distinct domains. It can easily handle customers whose needs a single organization can satisfy. However, setting up a project that requires contributions from several domains (i.e., subsystems) would be trickier. The project-driven approach given above is closer to how companies handle this situation today.

A company should also consider the intended use of information repositories when choosing organizational models. This is particularly important in considering how to evolve a domain. A domain changes based on customer needs, both real and anticipated. The organization responsible for controlling the domain has the responsibility of keeping abreast of such needs. In the project-driven model, the organization controlling a domain has no direct communication with customers. The organization must determine domain needs via the organizations that deal with the customers. This layer places some extra burden on the organization controlling a domain. By contrast, an organization using the domain-driven model has responsibility for both a domain and client projects, so customer needs feed directly into the organization.

4.9 THE APPROACH TO REUSE

Once an organization has analyzed a domain, and once it has implemented reusable components according to the resulting model, it must have a strategy by which projects can leverage the products. This strategy must be well-defined. Aside from guiding projects building applications in the domain, it should have guided the analysis and implementation of the domain. This point was made in Section 1.2, which states that reuse can be:

1. *Opportunistic.* The application implementor accesses a library with components designed to be reusable. The goal is to leverage existing software assets. The implementor has the responsibility of identifying places where reuse is possible, of locating components that fit the needs (or he can adapted to fit the needs), and of obtaining and integrating them.

2. *Systematic.* Reuse becomes part of a process that incorporates knowledge of how and when to reuse software within a domain. Systematic reuse also leverages existing software assets. However, the more important goal is to leverage future software efforts by devoting time up front to creating a suitable process.

(The choices omit ad-hoc reuse since it entails no formal domain analysis.) Studying this criterion involves considering when the real work in reusing software takes place. Under opportunistic reuse, the reuse of software (as opposed to building reusable components) occurs during application development. Developers must identify the need and potential for reuse. They must then identify, obtain, and tailor reusable components. Note that the developers are performing what might be termed "reuse operations" to locate and incorporate parts.

Under systematic reuse, most reuse operations occur during domain implementation. Domain developers build reusable software then. They also create reuse operations—products that define how to identify, obtain, and tailor components. An application developer uses these products to do mechanically what an application developer practicing opportunistic reuse has to design and implement.

Therefore, systematic reuse pushes most reuse operations out of application development and into domain development. It reduces the time to implement a given application when compared to opportunistic reuse, at the expense of time that domain analysts must devote to determining how they can systematized reuse. Unfortunately, no one has any data yet from which to determine a break-even point.

4.10 THE PRIMARY PRODUCT OF DOMAIN DEVELOPMENT

A remaining issue is the primary product that results from implementing a model of the domain. Here, "primary" means the product most visible to those intending to reuse software in a domain. The approaches offer the following possibilities:

1. ***Reuse Library.*** In all approaches, domain developers use the specification of the domain to build a reuse library. Often, it is what developers interact with when building applications in the domain. Reuse is opportunistic, occurring through adaptation of a canonical part, which might be implementation, design, or requirements.

2. ***Application Engineering Process.*** This is a process that leverages the information in a reuse library. See Section 2.2.

An organization can evaluate which is most appropriate based on many factors. Consider software process needs. A reuse library is potentially useful in conjunction with any process. It is more widely applicable than an application engineering process; an organization would have to institute shifts to accommodate the software process. Similarly, suppose an organization's intended use of its information repositories include making components generally available to all its application developers, expecting them to be able to modify the components quickly and simply. It can do so using a reuse library (if it has access to a suitable existing software base). However, if its goals include systematic reuse of parts, a reuse library will not be sufficient. A library contains parts but no systematic process for adapting and composing them.

5. APPLYING THE CRITERIA

This section applies the ten criteria to the six domain analysis approaches introduced in Section 2. It shows how particular methods and features of an approach correlate to resolutions of decisions for the criteria. This can help an organization choose an approach. The previous section showed how resolutions of each criterion relate to the contextual factors in Section 1.3. If an organization understands its needs in terms of those contextual factors—and given the factors, this seems quite likely—it can use the results of the analysis to determine if an approach is suitable.

Table 5 summarizes applying the comparison criteria to the six approaches introduced in Section 2. The rows list the criteria and the columns the methods. Each cell is a value for a criterion as applied to the method in the corresponding column. The table reveals that no two approaches are exactly alike, although some are more similar than others. This is no surprise, given the genealogy of Figure 1, but each approach has captured its own unique combination of decisions as to the optimal technique for analyzing, capturing, and using information about a domain.

Table 5. Summary of Approaches

	Method					
	Jaworski	**Synthesis**	**Prieto-Díaz**	**FODA**	**Lubars**	**KAPTUR**
Definition of "domain"	Business area	Business area	Application area	Application area	Application area	Application area
Determination of problems in the domain	Problem-oriented	Problem-oriented	Problem/ solution combination	Problem-oriented	Solution-oriented	Solution-oriented
Permanence of domain analysis results	Permanent	Mutable	Permanent			Mutable
Relation to the software development process	Pre-requirements, independent	Meta-process	Pre-requirements, independent			Meta-process
Focus of analysis	Objects and operations	Decisions	Objects and operations	Decisions	Objects and operations	Decisions, objects and operations
Paradigm of problem space models	Generic requirements	Decision model	Generic requirements	Decision model and generic requirements	Generic requirements	Generic requirements

Table 5, continued

	Method					
	Jaworski	Synthesis	Prieto-Díaz	FODA	Lubars	KAPTUR
Purpose and nature of domain models	Repository of domain knowledge	Process specification	Software specification			Process specification
Organizational model of domains and projects	Not specified	Domain-driven	Not specified			
Approach to reuse	Systematic	Systematic	Opportunistic	Opportunistic	Opportunistic	Systematic
Primary product of domain development	Reuse library	Application engineering process	Reuse library			

5.1 ANALYSIS OF CRITERIA

5.1.1 The Definition of "Domain"

Jaworski and Synthesis define domains as business areas. Jaworski's domains include control systems, signal processing systems, and command and control systems. In such domains, reasons for pursuing one approach or technology over another are driven mainly by economic and systems engineering factors. In fact, the principal determination of a domain is "whether software artifacts developed for one instance of the domain may be cost-effectively reused for another instance" (Jaworski et al. 1990). Furthermore, much of the early analysis (feasibility analysis) focuses on such concerns.

These statements also hold true for Synthesis. Moreover, Synthesis activities for domain evolution use customer needs as inputs, further strengthening the business area view.

Lubars, FODA, Prieto-Díaz, and KAPTUR define domains as application areas. Lubars defines domain analysis as the activity of "analyzing an application domain for reusability" (Lubars 1991, 163). His approach focuses mainly on those areas that can be decomposed into reusable design modules. His concerns are software-oriented and deal with developers' perspectives. His objective is to support software construction by composition of reusable components.

FODA defines a domain simply as "a set of applications" (Kang et al. 1990, 2). The inputs to a feature-oriented domain analysis are primarily functional in nature (environment constraints, software requirements); the process does not state how to relate them to business needs. The outputs are views of software from different perspectives: end-user, requirements analyst, and software designer/implementor.

KAPTUR says domain analysis is "the formulation of the common elements and structure of a domain of applications" (Moore and Bailin 1991, 179). In KAPTUR's supply-and-demand model, the demands for domain products come directly from applications rather than from the business forces that drive those applications.

Prieto-Díaz's definition on page 23 of this report also adheres to the application view. Both the top-down and bottom-up analyses concentrate on existing applications (although from different perspectives) rather than on external factors.

5.1.2 The Determination of Problems in the Domain

FODA follows a top-down approach to domain analysis, stemming from a stated set of problems; the analyst searches for generic solutions. FODA is therefore problem-oriented. Jaworski's approach has the analyst perform an OOA to define generic requirements. Defining the requirements ends domain analysis; during domain implementation, software developers define generic architectures. Therefore, Jaworski's approach not only advocates determining problems first, but the solutions are not even derived as part of domain analysis proper. Both Synthesis and Jaworski have the analyst focus initially on customer needs, i.e., the problems the customer faces. The analyst formalizes these problems, then creates solutions for them. In Synthesis, this process occurs iteratively. That is, the analyst defines a problem and specifies a solution. Based on the knowledge gained from this, he refines the problem and solution. However, he bases the description of the solution on the problem as stated so far; therefore, the focus is on determining problems first.

KAPTUR follows a bottom-up approach by focusing on existing systems which represent specific solutions. Analysts using KAPTUR must determine how they can generalize a set of solutions to solve a domain problem. KAPTUR is solution-oriented. This is also true of Lubars' approach, where the process for domain analysis consists of several stages of analysis of existing systems followed by abstraction of those systems' properties.

Prieto-Díaz uses separate analysis processes for different domain entities. He uses top-down analysis for high-level designs and requirements, and bottom-up analysis for low-level requirements and source code. The top-down analysis yields canonical structures and generic models. From these, people derive appropriate solutions that fit the models. Therefore, the top-down analysis is problem-oriented. The bottom-up analysis yields abstractions of existing applications. These are then matched to problems in the domain. Therefore, Prieto-Díaz's approach uses a problem/solution combination for problem determination.

5.1.3 The Permanence of Domain Analysis Results

In Synthesis, an organization establishes a software development process based on the results of domain analysis. As application developers build new systems, they feed changing or previously unrecognized customer and process needs back to domain analysts, providing for evolution of the domain analysis products. Synthesis domain analysis results are therefore mutable.

KAPTUR's creators explicitly state that "domain evolution **must** be considered in the analysis process" (Moore and Bailin 1991, 194). KAPTUR's supply-and-demand model supports this view in a manner similar to Synthesis. KAPTUR analysts, however, concentrate mainly on how reusable artifacts should evolve to meet changing domain needs. Synthesis considers process evolution as well.

Prieto-Díaz does not show mutability explicitly in his process model. However, he states that iteration is implicit in domain analysis. He contends that the results of the activities in his process provide feedback to subsequent iterations of the process. He therefore intends his results to be mutable.

In FODA, on the other hand, once a domain analyst defines a set of domain models (i.e., views), they remain very much fixed since they represent generic features in an application area. These models act as frames of reference for instantiations of new applications. Changing them could produce incompatibilities for reuse of existing components. FODA's results are permanent. This view also characterizes Jaworski, where the goal is to produce a "stable requirements framework." That is, the solutions to the problems may change, but the problems do not. Lubars also takes this approach. He intends that the abstractions created as part of domain analysis are fixed for a domain. The supporting products in his domain engineering activity change in response to technology but always conform to the specifications set forth by the domain analysis results.

5.1.4 The Relation to the Software Development Process

Jaworski's process places domain analysis activities in the system requirements, and to a lesser degree software requirements, phases of a waterfall model. The OOA techniques yield canonical requirements; the domain implementation activities yield canonical designs. Application developers are to use these products in the requirements and design phases, respectively, and domain analysts must complete them prior to those phases. However, Jaworski does not require them to be used in conjunction with a particular model (although he shows a mapping from domain analysis activities to a 2167A software process). His approach is therefore pre-requirements, independent.

FODA, Lubars, and Prieto-Díaz define processes but do not map them to software process models for application development. Instead, they suggest general strategies for using the products of domain analysis. For example, Lubars' approach and FODA both yield generic architectures. Implementors use them in software design (they create application-specific structures from the generic ones), but the domain analysis approaches do not prescribe properties of the design process.

Prieto-Díaz goes a little farther than FODA or Lubars. For each product of his approach, he suggests uses within software process phases. He does not tie his approach to a particular process model. He only shows how domain analysis products might be used. Organizations therefore have guidance on how to integrate domain analysis activities into their existing processes.

KAPTUR's supply-and-demand model assumes concurrent life cycles for domain engineering and application development. The cycles are independent, except to the degree that application needs determine the relative importance of domain analysis products. Moreover, the generic architectures that result from KAPTUR domain implementation include recommended processes for an application developer to use in tailoring the architectures to meet his application's requirements. The KAPTUR domain analysis process is therefore a meta-process.

The Synthesis domain analysis process is also a meta-process, but Synthesis goes further than KAPTUR. KAPTUR intends the recommended processes to fit into an organization's application development process. For example, an organization using a waterfall model might fit domain-specific processes into its design phase. In Synthesis, domain engineering yields the application development process, which incorporates existing organizational processes as appropriate. For instance, the application development process sometimes requires extensions to accommodate nuances of applications that contain features outside the domain in question. The process for specifying and implementing these nuances is organization-specific, not domain-specific. However, in Synthesis, the application development process provides the context for these other processes. KAPTUR incorporates the processes into an organization's existing model.

Synthesis and KAPTUR share another property that makes their domain analysis processes be meta-processes. The domain analysis activities specify certain required outputs (requirements specifications, software designs, etc.) but do not prescribe a process for generating these outputs. An organization first opts to perform domain analysis using either approach, then tailors the domain analysis approach to use its chosen methods (e.g., structured analysis, object-oriented design). Here again the activities for generating products are subordinate to the process for domain analysis.

No approach is "pre-requirements, dependent." Interestingly, this was not always true. Lubars originally defined his approach to be useful mainly for an OOA and design process. He subsequently modified his products so the abstractions are not in an object-oriented framework.

5.1.5 The Focus of Analysis

Feature-oriented domain analysis uses a single structure to describe both commonalities and variations. Generating this structure is one of the early tasks of domain analysis in FODA. Although the structure shows both commonalities and variations, its purpose is to help differentiate among the applications in the domain. FODA's creators chose a hierarchical model for this because they felt it was a simple structure for representing decisions about how applications differ. In FODA, then, the focus of analysis is on decisions that application developers make. FODA's creators were interested in capturing when the decision must be made—compile-time, load-time, or run-time—and created a decision model that reflects this information.

Synthesis' focus of analysis is also on decisions. The products of a Synthesis domain analysis reflect this: the domain specification contains process requirements, which are an initial statement of a decision process for a domain that application developers will follow. These decisions come from a decision model derived during domain analysis. The decisions in this model together define all applications in the domain. In Synthesis, unlike FODA, the decision model is separate from the product requirements (i.e., the statement of commonalities). Also, the products of Synthesis not only specify possible decisions, they describe how application developers can resolve decisions. However, the Synthesis decision model does not explicitly differentiate among the times that application developers resolve decisions. There is no way to indicate that a decision will be made at run-time, as there is in FODA's model. Synthesis is concerned only with pre-runtime decisions. In Synthesis, these are the only decisions that application developers can control. In FODA, application developers need to make design choices based on run-time decisions; they resolve the equivalent choices during domain engineering in Synthesis.

Jaworski's approach uses OOA techniques, and the first activity is to identify objects and operations shared by systems in the domain. Domain analysts capture the decisions that application developers make by making the products generic. However, the domain analyst concentrates on examining existing applications (and anticipated systems) for commonality and abstracting the characteristics of those applications. Prieto-Díaz and Lubars' approach are similar. In particular, Lubars' process is a series of "examine-and-abstract" phases. Jaworski, however, concentrates on requirements. Lubars concentrates on design, and Prieto-Díaz on design and code.

KAPTUR blends these two approaches. Domain analysts using KAPTUR follow an "examine-and-abstract" paradigm, but they also are expected to formulate a decision model for application developers based on their abstractions.

5.1.6 The Paradigm of Problem Space Models

In Synthesis, reuse depends strongly on capturing a solution to a problem in a domain in terms of how that solution differs from all others in the domain. This is expressed in terms of the decision model. The decision model, which is based on problem space concepts, therefore becomes the paradigm.

Prieto-Díaz, on the other hand, assumes reuse stems from a realization that a problem has an existing solution, or something close, already implemented. An implementor must first recognize the general problem, find a matching general solution, then tailor that solution. Generic, reusable requirements are therefore important; describing variations among these requirements is secondary.

Jaworski also uses generic requirements as the paradigm for problem space models. Indeed, his requirements (derived through OOA) are true requirements for the domain. Prieto-Díaz's are specifications for design objects. Jaworski does not advocate building reusable designs until domain implementation.

In Lubars' approach, the results of each process phase are abstractions of the objects and operations studied in the phase. Domain analysts express the abstractions as data flow diagrams, showing functions that exist in a domain. The problem space model therefore shows generic requirements in terms of functions available.

In FODA, although the focus of analysis is on decisions, the products of domain analysis that emphasize a problem space model consist of a features model, an ERA model, a data flow model, and an FSM model. The features model indicates a decision-oriented problem space model paradigm. The other three products, however, are generic, reusable specifications of various properties of a domain. Furthermore, none of the four products are really **the** paradigm, because each model provides a different paradigm to a particular class of people who use the products of domain analysis—end-users, developers, and requirements analysts, respectively (requirements analysts use the last two products). The paradigm of the problem space model therefore depends on who is using it. This is an emphasis of both decisions and generic requirements.

KAPTUR, like FODA, uses multiple models to represent different views of domains, and these models are generic. KAPTUR's developers go to great lengths to justify their belief that these semi-formal models are better than fully formal ones. They base their paradigm of reuse on examining these models to locate one that approximately matches requirements, then tailoring the part using a decision model of "distinctive features" that they have developed for each model. However, unlike FODA, the features are not separable from the models, and different user classes do not use them. KAPTUR's problem space model paradigm is therefore generic requirements.

5.1.7 The Purpose and Nature of Domain Models

Jaworski advocates creating a knowledge base containing all known facts about a domain, with an intelligent front-end to help analysts and engineers retrieve information. This is a repository of domain knowledge.

FODA domain models are design architectures that application implementors can refine into implementations; they therefore specify a class of software products, bounded by the architectural model. Lubars' domain models share this property, although they use different representation mechanisms. Whereas FODA domain models consist of four separate views (see Section 5.1.6), Lubars'

approach specifies abstract design schemas that domain developers refine into more specialized ones. A type lattice introduces constraints on the refinement process by specifying properties of the schemas. The lattice is first used during domain engineering (Lubars' term for what other approaches call domain implementation), not domain analysis, but Lubars considers it part of the domain model.

Prieto-Díaz's approach yields specifications of software products. The specifications are stated as abstract operations and as generic architecture descriptions. They are further organized by a faceted classification, structuring them for easy access.

KAPTUR, like FODA, lets domain analysts use several models to specify functional and architectural properties of software products. It also adds a process to the domain model that guides application developers as they refine the architecture. It is therefore a specification of both product and process. Synthesis goes one step farther by specifying product, process, and environment; the last consists of descriptions of tools that help application developers follow the process. Moreover, the person using this environment sees only decisions and not, as a rule, design structures or even functional specifications. That is, the application developer would not make a choice between two design structures. Instead, the domain analyst will have identified the conditions where each structure is appropriate. The application developer then decides which condition is relevant, and the process for application development includes information on how to map the condition onto a design structure. The intent is to bring the decision-making process up from the level of software concerns to the realm of how decisions relate to the environment in which application developers will use the software.

5.1.8 The Organizational Model of Domains and Projects

Synthesis includes a domain management activity. It provides for organizational control of a domain in the sense that the organization, and not a product, defines how the domain evolves. This implies a domain-driven organization. Actually, a project using Synthesis can use either of the other organizations; that is, Synthesis espouses a domain-driven organizational model but does not mandate it.

The other approaches to domain analysis do not discuss this matter explicitly. KAPTUR's developers plan to use it in a project-driven organization: they will support reuse in a project working in the SMEX (Small Explorer) domain.

No approach conforms strictly to one organizational model. As such, this criterion is not useful for determining which approach to use. The Consortium includes it because it feels it is an important consideration no matter which approach an organization chooses, and because it believes that, in the future, domain analysis approaches will define processes that favor one organization more strongly than another.

5.1.9 The Approach to Reuse

In Lubars' approach, domain analysts use his ROSE-1 tool to enter abstract designs in a reuse library. Application developers use ROSE-1 to search for existing designs that meet their needs. This is opportunistic reuse. They search based on matching of data flow types and properties, plus keywords in function descriptions.

Prieto-Díaz also advocates opportunistic reuse, but with some differences. In his original approach, application developers searched only for code. His newer approach has provisions for generic designs, although without the refinement rules present in Lubars' approach. In both cases, his domain model organizes information according to a faceted classification, a simple but proven technique for locating information (Lubars claims that data flow properties are similar to faceted classification).

FODA's approach to reuse is also opportunistic. Application developers locate and refine design structures. The features model guides refinement, mapping features to parts of the generic models and thereby indicating whether they are present in a given application. FODA has no inherent mechanism for categorizing parts other than what the features model implies. FODA therefore is more appropriate as a basis for determining the design of an application (which was one of its goals) than as a means to locate individual parts within a domain (Prieto-Díaz's analysis seems better suited to this).

In Synthesis, domain analysis yields a decision model and an application development process based on the decision model. The decision model, in essence, describes solutions to problems in the domain; in other words, the range of expressible decisions is the set of all implemented solutions. The process of application development amounts to resolving decisions on which problem to solve. During domain analysis, the model is mapped to adaptations of components. This mapping lets application developers mechanically identify, obtain, and adapt components based on the information in the decision model. Synthesis therefore supports systematic reuse.

KAPTUR also supports systematic reuse, although in a different way. Synthesis systematizes reuse based on decisions that application developers need to make about applications in a domain. KAPTUR systematizes reuse in terms of design decisions. Domain analysis yields a mechanical process that application developers use for refining designs into implementations. The application developers must have already determined the appropriate design structure, a task that would be part of domain engineering in Synthesis, not application development.

Jaworski also advocated systematic reuse. Domain analysts would determine a process for transforming the generic requirements into implementations. Application developers would use this process by supplying values for the generic parameters of objects in a domain.

5.1.10 The Primary Product of Domain Development

All approaches except Synthesis have a reuse library as the primary product of domain development. However, the content and use of the library varies between approaches. Jaworski's information repository is the most general-purpose library. It contains any domain information deemed potentially relevant, a huge amount of material for a sizeable domain—but all potentially useful in the problem-solving process. The other approaches have more specific needs and place less material in the library. Prieto-Díaz's products center around what his indexing scheme can describe. The library supports parts retrieval. Lubars' approach and KAPTUR are both built around automated environments. Their libraries intend to support not only retrieval but refinement. FODA's creators had this in mind as well, although they have not yet treated automation in detail.

In Synthesis, a reuse library is one product, but the application engineering process defines how to use that library. It therefore shapes the library, rather than the other way around, and is the primary product.

5.2 BENEFIT OF THE COMPARISON CRITERIA

A motivation for this study has been to analyze the feasibility of a unified approach to domain analysis applicable across domains and across organizations. This analysis led to the identification of several criteria for comparing the approaches which organizations can use not only to assess the possibility of a unified method, but, as stated in the introduction, to help practitioners discover which approach best meets their needs.

The practical benefit of this result is of utmost importance to any organization planning or currently implementing a reuse program. The ability to characterize a domain analysis method for selection based on the needs and goals of an organization is a prerequisite to establishing a reuse-centered software development process. Domain analysis products are key components in this process. A reuse library, for example, is essential for reusing components, and an architecture model is essential for guiding the composition of new systems. The selection of the proper domain analysis process can determine the success or failure of a reuse program.

The Consortium selected the criteria from what it considered to be the basic characteristics of the domain analysis approaches surveyed. It placed special emphasis on the selection of the value names assigned to distinct attributes to facilitate their association to specific organization requirements. The criterion for "purpose and nature of domain analysis" has the values repository, software specification, and process specification. If an organization is, for example, at an SEI maturity level 4 (managed) (Humphrey 1989), then a domain analysis process that focuses on process specification will be more desirable for their reuse program than one whose focus is on a repository.

Although an ideal outcome of this study would be a set of guidelines on how to apply these criteria to given organization requirements, objectives, maturity level, resources, and the like, the criteria presented are generic enough to be applicable without guidelines. A further effort should concentrate on characterizing organization attributes and propose guidelines to map those attributes to the comparison criteria to provide a systematic and more formal selection process.

This page intentionally left blank.

6. CONCLUSIONS

This report concludes by discussing what the Consortium learned from this study, in terms of the four goals given in the introduction.

6.1 IS A UNIFIED DOMAIN ANALYSIS APPROACH FEASIBLE?

Using a unified domain analysis approach implies that one believes the approach can meet a broad range of analysis needs. The approach should accommodate all permutations of the contextual factors discussed in Section 1.3, or at least all those relevant to an organization. Thus the approach must be responsive to the variety of business needs, the evolving software process, the changes in domain knowledge, etc., that are all part of an organization's infrastructure.

It seems clear that, among the approaches surveyed, no approach is "best." One approach may be better suited to a given organization's needs than any other (see Section 6.2), but organizations have widely different goals. No existing approach seems able to satisfy so broad a range of goals that the Consortium could call it better than others. Therefore, a unified domain analysis approach must not be an existing approach, but some amalgamation of concepts from approaches.

The Consortium believes that a unified approach is not feasible or even desirable. The problem, in brief, is that a unified approach assumes that the ways information is obtained and used depend more on the approach than on the domain and the organization using it. Experience strongly suggests that the reverse is true. Library scientists advocate faceted classification because experiments have shown its efficacy in organizing information. Therefore, a unified domain analysis approach must yield a faceted classification if an organization interested in categorizing parts for direct look-up is ever to use it. Some organizations will certainly want this, but others may not care. The domain analysts, unfortunately, pay the price of having to decide which features of the approach to use and which to ignore. In short, although some criteria in Section 4 are related, there are many possible combinations of characteristics. To place all of them in a single, unified approach would complicate that approach to the point of incomprehensibility.

This conclusion is perhaps no surprise. No requirements specification notation, design method, programming language, or CASE tool has proven to be best, or even well-suited to all areas. There is no reason to suppose that a single domain analysis approach would either. An important area of future research will be learning how to tailor approaches to specific ends.

6.2 SELECTING THE RIGHT DOMAIN ANALYSIS APPROACH

Assuming there is no point in creating a unified approach, practitioners must continue to select from among existing ones. An organization can base its selection on the context in which it will use a domain analysis process. The factors that make up this context are things that organizations will want to

analyze in any case to improve their software productivity and to create useful, marketable products. The Consortium therefore feels justified in basing the criteria on them.

The selection process is still qualitative, not quantitative. Quantifying certain contextual factors is difficult. It is hard to measure the state of domain knowledge, for instance, or to state business objectives precisely enough to relate them directly to software needs. However, organizations can use the criteria to reject approaches that are wrong for their objectives. The right approach may then come down to such mundane factors as availability of automated support in the organization's environment or to personal tastes in dogmatic areas (e.g., whether domain analysts prefer object-oriented approaches).

The criteria are not intended as final, but rather based on the current state of the art. The Consortium will continue to refine them as our understanding of domain analysis expands.

6.3 TRENDS IN DOMAIN ANALYSIS RESEARCH

Domain analysis researchers have shifted their emphasis from code reuse to reuse of more abstract structures. Several years ago, Booch's Ada parts (Booch 1987) and the Common Ada Missile Packages (CAMP 1987) were state-of-the-art domain analysis products. Now the trend is to have analysts concentrate on design architectures, requirements and document components, and other products that map, manually or automatically, to design. Domain analysis does not ignore code reuse, but now it is seldom the starting point for analysis, nor the most visible end product.

This trend probably reflects increased confidence in the ability to analyze domains. This report has argued that domain analysis for code reuse is in some sense the easiest kind, but it also has the least potential payoff. The trend toward the study of requirements and design structures indicates researchers now believe that people can create useful abstractions of such structures, which have a higher payoff. Adopting this view will require a more encompassing view of reuse than is usual today. Just what is meant by reusing a design is still not widely accepted.

6.4 COMPARING AND CONTRASTING APPROACHES TO DOMAIN ANALYSIS

A key benefit of this study is a framework for comparing and contrasting domain analysis approaches. The criteria presented are an initial set that need further refinement and expansion. They can evolve, eventually, into a set of formal guidelines, not just for systematically selecting domain analysis methods, but for guiding the development of new methods aimed at meeting specific criteria. Organizations would customize these new domain analysis methods to support individual needs.

REFERENCES

Arango, Guillermo
1988
Domain Engineering for Software Reuse. Ph.D. Dissertation. Irvine, California: University of California, Department of Computer Science.

Bailin, Sidney, and John Moore
1989
The KAPTUR Environment: An Operations Concept. Rockville, Maryland: CTA Incorporated.

Basili, Victor,
H. Dieter Rombach, J. Bailey,
and B. Joo
1987
Software Reuse: A Framework. *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse.* Blue Mountain Lake, New York.

Bollinger, Terry, and
Clem McGowan
1991
A Critical Look at Software Capability Evaluations. *IEEE Software* 8, 2:25–41 (July).

Booch, Grady
1987
Software Components with Ada. Menlo Park, California: Benjamin/Cummings, Inc.

Burkhard, Neil, Jeff Facemire,
James Kirby, and
James O'Connor
1990
Applying Synthesis in the Design Composer Domain. SPC-90078-MC. Herndon, Virginia: Software Productivity Consortium.

CAMP
1987
Common Ada Missile Packages. Technical Report, Product Presentation Literature. St. Louis, Missouri: McDonnell-Douglas Corporation.

Coad, Peter
1989
OOA—Object-Oriented Analysis. Austin, Texas: Object International, Inc.

Cruickshank, Robert, and
John Gaffney
1991
The Economics of Software Reuse, SPC-91128-MC. Herndon, Virginia: Software Productivity Consortium.

Devanbu, Premkumar, Ronald
Brachman, Peter Selfridge, and
Bruce Ballard
1991
LaSSIE: a Knowledge-based Software Information System. In *Domain Analysis and Software Systems Modeling.* Edited by R. Prieto-Díaz and G. Arango, 150–162. Los Alamitos, California: IEEE Computer Society Press.

Frakes, William, and Paul Gandel 1987 — Classification, Storage and Retrieval of Reusable Components. *Proc. 20th Annual HICSS*, 530–535. Kona, Hawaii.

Gomaa, Hassan 1984 — A Software Design Method for Real-Time Systems. *Communications of the ACM* 27:938–949.

Humphrey, Watts 1989 — *Managing the Software Process.* Menlo Park, California: Addison-Wesley.

Jaworski, Alan, Fred Hills, Tom Durek, Stuart Faulk, and John Gaffney 1990 — *A Domain Analysis Process.* DOMAIN_ANALYSIS-90001-N. Herndon, Virginia: Software Productivity Consortium.

Kang, Kyo, Sholom Cohen, James Hess, William Novak, and Spencer Peterson 1990 — *Feature-Oriented Domain Analysis (FODA) Feasibility Study,* CMU/SEI-90-TR-21. Pittsburgh, Pennsylvania: Software Engineering Institute, Carnegie-Mellon University.

Lubars, Mitchell 1987 — A Knowledge-Based Design Aid for the Construction of Software Systems. Ph.D. Dissertation. Urbana, Illinois: Department of Computer Science, University of Illinois.

1991 — Domain Analysis and Domain Engineering in IDeA. In *Domain Analysis and Software Systems Modeling.* Edited by R. Prieto-Díaz and G. Arango, 163–178. Los Alamitos, California: IEEE Computer Society Press.

Lubars, Mitchell, and M. Harandi 1987 — Knowledge-Based Software Design Using Design Schemas. In *Proceedings of the Ninth International Conference on Software Engineering,* 253–262. Monterey, California.

Moore, John, and Sidney Bailin 1991 — Domain Analysis: Framework for Reuse. In *Domain Analysis and Software Systems Modeling.* Edited by R. Prieto-Díaz and G. Arango. 179–203. Los Alamitos, California: IEEE Computer Society Press.

Neighbors, James 1984 — The DRACO Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering* SE-10:564–573.

Parnas, David 1976 — On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2, 1:1-9.

Parnas, David, Paul Clements, and David Weiss 1985 — The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* SE-11:259–266.

Prieto-Díaz, Rubén
1987

Domain Analysis for Reusability. *Proceedings of COMPSAC'87*, 23–29. Tokyo, Japan.

1990

Domain Analysis: An Introduction. *Software Engineering Notes* 15, 2:47–54.

1991

Reuse Library Process Model. Final Report for STARS Reuse Library Program, Contract F1962-88-D-0032. Hanscom Air Force Base, Massachusetts: Electronics Systems Division, Air Force Systems Command.

Prieto-Díaz, Rubén, and Guillermo Arango
1991

Domain Analysis and Software Systems Modeling. Los Alamitos, California: IEEE Computer Society Press.

Ritchie, Dennis, and Ken Thompson
1974

The UNIX Time-Sharing System. *Comm. ACM* 17, 7:365–375.

Shlaer, Sally, and Stephen Mellor
1989

An Object-Oriented Approach to Domain Analysis. *Software Engineering Notes* 14:66–77.

Snodgrass, Jerry, Ted Davis, Neil Burkhard, Guy Cox, Jeff Facemire, Fred Hills, and James O'Connor
1990

Synthesis: Status and Results of Studies. SYNTHESIS_STUDIES-90041-P. Herndon, Virginia: Software Productivity Consortium.

Software Productivity Consortium
1991

Synthesis Guidebook, SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium.

Weiss, David
1990

Synthesis Operational Scenarios. SYNTHESIS_OP_SCENARIOS-90038-N. Herndon, Virginia: Software Productivity Consortium.

This page intentionally left blank.