DOMAIN ANALYSIS: AN INTRODUCTION

Rubén Prieto-Díaz
The Contel Technology Center
Fairfax, VA 22021

ABSTRACT

The objective of this paper is to provide a brief introduction to the area of domain analysis as seen from the software engineering perspective. The approach is by illustrating the concepts through selected reported experiences and to point out the specific characteristics of these experiences that relate to domain analysis. Definitions are introduced after the examples to avoid over explaining the concepts. A model for the domain analysis process is also proposed. The concept of a library based domain infrastructure is introduced as an attempt to show how domain analysis is integrated into the software development process.

A second objective in this paper is to give a perspective on some of the research issues facing domain analysis. The nature of the process calls for a variety of multidiciplinary issues ranging from knowledge acquisition and knowledge representation to management and methodologies to cultural and social questions.

1. Introduction

We define domain analysis as a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems. During software development, information of several kinds is generated. From requirements analysis to specific designs to source code. Source code is at the lowest level of abstraction and is considered the most detailed representation of a software system. Complementary key information is also generated during software development. Code documentation, history of design decisions, testing plans, and user manuals are all essential to convey a better understanding of the total system.

One of the objectives of domain analysis is to make all that information readily available. In making a reusability decision, that is, in trying to decide whether or not to reuse a component, a software engineer has to understand the context which prompted the original designer to build the component the way it is. The chain of design decisions used in the development process are absent in the source code. By making this development information available, a reuser has leverage in making reuse more effective.

A more dramatic improvement of the reuse process results when we succeed, through domain analysis in deriving common architectures, generic models or specialized languages that substantially leverage the software development process in a specific problem area. How do we find these architectures or languages? It is by identifying features common to a domain of applications, selecting and abstracting the objects and operations that characterize those features, and creating procedures that automate those operations. This intelligence-intensive activity results, typically, after several of the "same kind" systems have been constructed. It is then decided to isolate, encapsulate, and standardize certain recurring operations. This is the very process of domain analysis: identifying and structuring information for reusability.

Unfortunately, domain analysis is conducted in an ad-hoc manner and success stories are more the exception than the rule. The process of concept abstraction from identifying common features is usually considered as an exclusive human (i.e., intelligent) activity and commonly associated with "experience". Expert programmers, for example, are more proficient in coming up with the appropriate program construct that solves a given programming problem than novice programmers. Through experience, experts have created a larger collection of abstracted templates they can draw from when trying to solve a problem. This is reuse of encapsulated knowledge. Little is known about the process involved in deriving and organizing such collections of abstract concepts. Gaining experience is a slow unstructured learning process. Similarly, domain analysis is a slow unstructured learning process that leads to the identification, abstraction, and encapsulation of objects in a particular domain.

Typically, knowledge of a domain evolves naturally over time until enough experience has been accumulated and several systems have been implemented that generic abstractions can be isolated and reused. Take for example the domain of report generators. It was recognized that report generation was an inherent component of all business applications systems. Experts analyzed the basic functions involved in generating reports and designed systems to automate the process. A firing neuron model may be appropriate to illustrate the domain analysis process; it requires a critical charge to achieve a qualitative change. In domain analysis, experience and knowledge is accumulated until it reaches a threshold. This threshold can be defined as the point when an abstraction can be synthesized and made available for reuse.

The current ad-hoc nature of domain analysis can be compared to the way software was developed in the early days. Cleverness was the rule in the individually developed specialized programs of the time. As the need for larger more complex systems emerged, more systematic and structured approaches were developed. Cleverness is now the rule in analyzing a domain. Success translates into identifying the right domain, abstracting the essential objects and operations, encapsulating them in the form of procedures or generic architectures or a formal language, and reusing them. Examples range from spreadsheets to forms management to small specialized languages (e.g., numerical control machines, SQL). In order to truly exploit reusability in more complex domains we need to develop formal approaches to domain analysis. A goal in domain analysis research is to provide the means to facilitate charging the neuron of our model for the qualitative change required for a high level reuse. To accomplish this goal we must find ways to extract, organize, represent, manipulate and understand reusable information, to formalize the domain analysis process, and to develop technologies and tools to support it.

Domain analysis research can benefit significantly from work in other areas. Many of the issues in domain analysis are also issues in other disciplines. Reusable information is a kind of knowledge and extensive research and technologies are available. Artificial intelligence offers tools and techniques for knowledge acquisition and knowledge representation. Systems analysis also offers well-developed techniques and proven methodologies to help us understand the domain analysis process. Information management technology such as hypertext may bring ideas in visualizing and understanding reuse information. Domain analysis research should try to reuse existing research from other disciplines.

2. Background

The term *domain analysis* was first introduced by Neighbors [Nei81] as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain." He draws the analogy of domain analysis to systems analysis. The difference being that systems analysis is concerned with the specific actions in a specific system while domain analysis is concerned with actions and objects in all systems in an application area. During his research with Draco, a code generator system that works by integrating reusable components, he pointed out that "the key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code."

Neighbors later introduced the concept of "domain analyst" [Nei84] as the person responsible for conducting domain analysis. The domain analyst plays a central role in developing reusable components. A domain analyst is expected to be a person of all trades. He or she must understand systems analysis, the domain of application, the software technology at hand, and be able to communicate with the players in each of these areas. To better meet all these requirements, Neighbors goes further in suggesting that the domain analyst be also a domain expert.

The Common Ada Missile Packages (CAMP) Project [Cam87] took Neighbors' ideas into practice. The CAMP Project is the first explicitly reported domain analysis experience and they acknowledge that "[domain analysis] is the most difficult part of establishing a software reusability program". They analyzed eleven tactical missile systems, identified several common components, and grouped them by their functionality. A set of general design templates in the form of Ada generics were derived and later integrated in a design support system, the Ada Missile Parts Engineering Expert (AMPEE), that supports component identification, component selection, and component construction.

Neither Neighbors nor the CAMP project address the issue of "how to do" domain analysis. Both focus on the outcome, not on the process. McCain [McC85] makes an initial attempt at addressing this issue by integrating the concept of domain analysis into the software development process. He proposes a "conventional product development model" as the basis for a methodology to construct reusable components. Market analysis is the first step in this product-oriented construction paradigm. Once the marketplace has been identified, the central activity becomes domain analysis. The main concern in this approach is "how to accommodate the largest number of potential product users in the product marketplace." In other words, how to identify, a priori, the areas of maximum reuse in a software application. McCain introduces a set of guidelines to conduct domain analysis. The process has three basic steps that recur repeatedly for different kinds of components:

1 identification of reusable entities
2 abstraction or generalization
3 classification and cataloging for further reuse.

This product-oriented paradigm has been tested successfully in several projects at IBM Federal Systems. Their experience in this approach was an important factor for a recent award as principal contractor in the STARS (Software Technology for Adaptable and Reliable Systems) project.

Drawing in part from the above experiences, Prieto-Díaz [Pri87] proposed a more cohesive procedural model for domain analysis. His model is based on a methodology for deriving specialized classification schemes in library science. In deriving a faceted classification scheme, the objective is to create and structure a controlled vocabulary that is standard not only for classifying but also for describing titles in a domain specific collection. Prieto-Díaz extended this technique to domain analysis as a procedural model in a series of data flow diagrams. He defines specific activities and intermediate products that not only convey a better understanding of the domain analysis process but provide the basis for identifying tools that could support some of these intermediate activities. Another contribution of this model is the definition of inputs and outputs to the domain analysis process. A key control

input is a domain analysis methodology custom-built for each specific domain. Outputs include: domain model, domain taxonomy, domain language, domain standards, and reusable components. A project at GTE Laboratories is currently underway to test this model.

In an analysis of the process of building application generators and its relation to domain analysis, Cleaveland [Cle88] points to the need for a methodological approach to building generators. The ad-hoc nature of building application generators often results from insights spontaneously gained on the job, rather than from the systematic application of methods and are, therefore, seldom documented. In an attempt to formalize the process, Cleaveland identifies seven basic steps for building application generators. All but one are "the job of the domain analyst." Key steps are recognizing domains, defining domain boundaries, and defining an underlying model. One of the most difficult tasks is recognizing "when a domain is mature for a generator". He suggests two heuristics: 1) has a formal or informal notation surfaced within the user community, and 2) are there identifiable patterns or regularities in the applications generated in a user community? A concern in domain analysis research is that we can not wait indefinitely for a natural evolution of a domain but must stimulate its maturation process through a systematic process.

Arango [Ara88] focused on this concern and proposed a different approach to domain analysis. The basic premise in this approach is to see reuse as a learning system. The software development process is seen as a self-improving system that draws from a "reuse infrastructure" as the knowledge source. Domain analysis is then a continuing process of creating and maintaining the reuse infrastructure. Arango breaks new ground in formalizing the process. Assuming the existence of a reuse infrastructure consisting of reusable resources and their descriptions (e.g., a repository) and given a specification of the system to be build, a reuse system attempts to produce an implementation of the specified system by reusing information from the infrastructure. In closing the process loop, the output (i.e., the implementation) is compared to the input and the system is evaluated by a measure of performance. In order to keep a good performance level, the infrastructure is continuously revised. Domain analysis is therefore integrated in the software development process. Arango's model is being tried at Kone Corporation, Finland, in the domain of elevators software.

The chronology just described deals with a line of work in software engineering explicitly called domain analysis. There are several research efforts in many other disciplines such as knowledge engineering and conceptual modeling that are concerned with similar problems and in the end achieve the same results. In many ways these efforts are identified with domain analysis. The difference is that the problem solving task being supported is not software engineering. To cover all ramifications of this research area are beyond the scope of this paper. In the paragraphs below we describe selected research efforts to provide a broader view of domain analysis.

3. Related Domain Analysis Experiences

ΦNIX- The late (phi)NIX project at Schlumberger-Doll Research [Bar85] is an example of a domain-specific automatic programming system. Specialized oil-well logging knowledge is captured and organized into problem solving rules that generate informal specifications. These informal specifications are further formalized through a refinement process that adds domain specific details. At this point domain independent rules that deal with programming implementation issues are applied to generate code. One of the objectives of this project was to demonstrate that automatic programming is more effective if there is a separation of domain knowledge from programming knowledge. One of their conclusions was that there is a need to know more "about how to organize and structure domain knowledge" so that it can be reused. They note further that solving these kinds of problems is what would make possible for "automatic programming systems to learn from experience."

We presume, since it has not been reported otherwise, that domain analysis in the (phi)NIX project was conducted informally. Expert knowledge from petroleum engineers, geophysicists, and geologists was used to define two specific software domains with high potential for reuse. Refinements during knowledge acquisition and discovery of common processes led to the proposal of automatic programming templates. A goal in domain analysis would be to make this process formal, or at least systematic.

Programmer's Apprentice- Another research effort where domain analysis has become a key factor is the Programmer's Apprentice Project [RW88], a system based on the concept of assisting the programmer in the routine details of programming analogous to how chief programmer teams work. A key concept in this project is that "programmers seldom think in terms of primitive elements" like low level programming instructions, but rather, they think "mostly in terms of commonly used combinations of elements". They call these familiar combinations "clichés." A cliché is a well defined unit of knowledge about a programing construct that can be abstracted, represented, and reused. A user of the programmer's apprentice assembles programs by inspecting clichés selected from a library.

Creating a library of clichés can be viewed as a process of domain analysis. A cliché must be identified first, typically through some commonality analysis where a programming construct is seen to recur several implementations. Next, the cliché is abstracted and refined so that it truly represents all implementations examined. Finally, it is encapsulated in a representational form that lends itself to reuse. They recognize the importance of this activity by noting that "codifying clichés is a central activity" in the project.

KATE- A step towards understanding the process of knowledge extraction and use in the software development process has been the KATE project [FN88]. The KATE project recognizes the limitations of current software specification techniques. In these techniques most of the knowledge on specification construction remains with one person; the expert. In trying to bring more of this type of expert analysis knowledge into the computer, they have proposed a system that "tries to acquire problem specification" for an intended system. They see the production of systems specifications as an interactive problem-solving process based on critique. Their analyst assistant (KATE) asks questions and issues critiques to a starting set of specifications. The interactive sessions between client and assistant is a refinement process to create valid, unambiguous, and consistent specifications. Since "clients have only a vague notion of what they want and only a narrow view of what is possible," a "good analyst has expertise in pulling out the key points of a client's problem."

The expertise of KATE has its origin in a set of protocols collected over several analysis sessions. KATE uses these protocols to elicit specification information from the client. Protocol collection, selection, analysis, and coding is an example of domain analysis. To be effective, however, KATE also requires knowledge of the application domain, knowledge of implementation possibilities, and knowledge of the production environment. Knowledge extraction, selection, and representation for each of these three kinds of knowledge are also examples of domain analysis.

A more significant contribution of the KATE project to domain analysis research is the concept of knowledge extraction by critique. An initial body of domain knowledge can be expanded by asking selected hypothetical questions in the form of critiques. By making these critiques domain independent, the process can be applied to any domain.

Library Science- Another application of domain analysis occurs in Library Science when deriving specialized classification schemes [Vic60]. Specialized faceted classification schemes are derived through a manual process that consists of grouping related terms from a sample of selected titles, defining facet names from such groups, ordering the terms in each facet, and specifying rules for synthesizing compounded classes. The resulting classification scheme becomes a conceptual model for the domain of the collection. Grouping of terms from titles is equivalent to finding objects and operations in an application domain. The naming of facets and defining classification rules is equivalent to deriving a domain model or creating a domain language.

Caster- Domain analysis is also evident when developing expert systems. Thompson and Clancey [TC86], developed their "Caster" expert system by reusing the MYCIN skeleton and found that the most crucial step in their experiment was domain analysis. They had to encapsulate all available knowledge in the sand-casting domain into a qualitative model. Based on 15 major classes of sand-casting malfunctions that cause shrinkage cavities, they used manuals, textbooks, and expert consultation to identify the abnormal processes that could cause shrinkage and then organized them into a classification hierarchy. "Only after we really understood the causality behind shrinks could we put that knowledge in the language used by our system." This process took the longest time.

OO- Object-oriented software development is another area where domain analysis is often applied. In object-oriented programming, identifying objects, operations, and their relationships in large systems is not as trivial a task as it is usually assumed. There is often the need to deal with objects at different levels of abstraction, define abstraction boundaries, find complex relationships, and find all possible operations performed on a given object [Boo86]. Moreover, there is the need to identify classes of objects and identify their common attributes. All these tasks, although not explicitly called domain analysis, fit our description of domain analysis.

Software Factories- Domain analysis is also an essential activity when setting up a software factory. The basis for the success of several software factories [LP79, Mat81, TM84, Mat87] has been the creation of libraries and catalogs of reusable components and the creation of common "logic structures" or "paradigms" for standardization. The standardization of design and programming methodologies have also been essential. These activities can be associated with the process of identifying, capturing, and organizing reusable resources.

As a concluding remark it is worth mentioning that the importance (and difficulty) of identifying, capturing, and organizing "appropriate" reusable resources has become clear as a larger segment of the software engineering community attempts to realize the promise of reusability. One of the recommendations from the 10th Minnowbrook Workshop on Software Reuse in July 1987 suggests "concentrating on specific applications and domains (as opposed to developing a general reusability environment)" [AM88]. The Rocky Mountain Workshop on Software Reuse [rmi87] acknowledged the lack of a theoretical or methodological framework for domain analysis. Since then, many other workshops (e.g., [bsh88], [oom88], [rip89]) have devoted a substantial amount of time to address the domain analysis issues directly.

4. Definitions

Domain: In a broad context it is "a sphere of activity or interest: field" [Webster]. 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, numerical control. Domains can be broad like banking or narrow like arithmetic operations. Broad do-

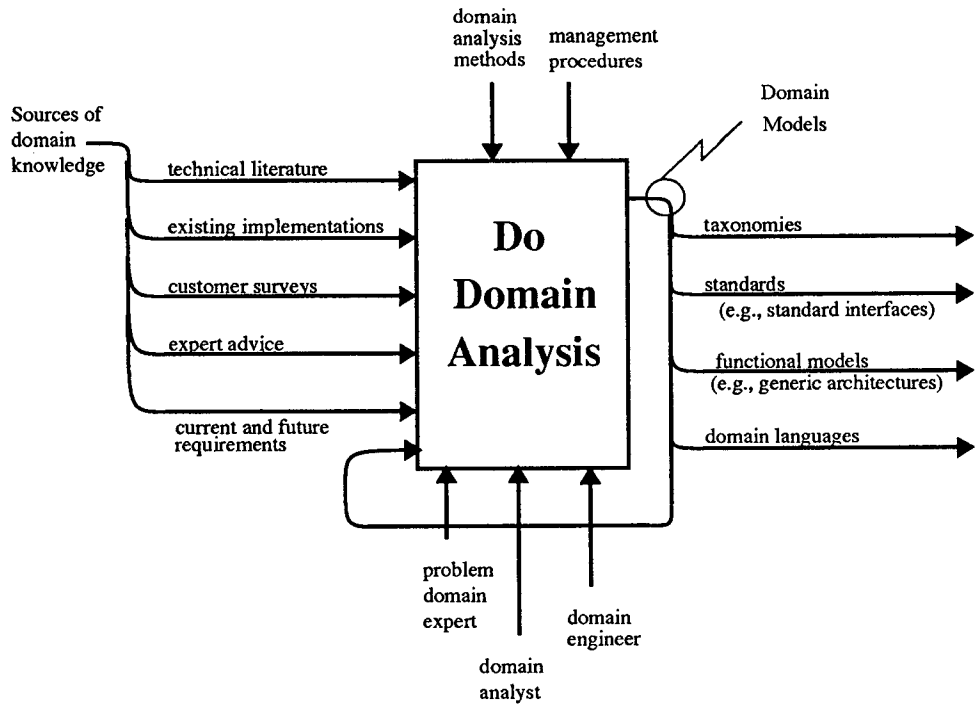

Figure 1. SADT Context View of Domain Analysis

mains consist of clusters of interrelated narrower domains usually structured in a directed graph.To "reserve a seat" in the domain of airline reservation systems, for example, an update operation is called from the domain of database systems. To "update a record" in the database domain, operations from a still more basic domain, like programming languages, are needed. Other domains like user interfaces (e.g., screen manipulation, mouse interaction) are also instrumental for airline reservation systems. Domains, therefore, can be seen as networks in some semihierarchical structure where primitive, narrow domains such as assembly language and arithmetic operations are at the bottom and broader, more complex domains are at the top. Domain complexity can be characterized by the number of interrelated domains they require to be operational.

Domain Boundary: Each domain in these domain networks is limited by a boundary that defines its scope. The borders define what objects, operations, and relationships belong to each domain and delimit their operational capability.

Domain Analysis: As indicated in the introduction above, domain analysis can be seen as a process where information used in developing software systems is identified, captured, structured, and organized for further reuse. More specifically, domain analysis deals with the development and evolution of an information infrastructure to support reuse. Components of this infrastructure include domain models, development standards, and repositories (libraries) of reusable components. Domain and boundary definitions are also activities of domain analysis. Unfortunately, a standard (universal) definition of domain analysis is yet to come. Due to the nature of the activities and issues involved and to the newness of the area, domain analysis is perceived differently by different communities. One of the objectives of this paper is to channel some of these differing perceptions towards a unified view of domain analysis.

5. The Domain Analysis Process

To better illustrate our view of domain analysis, the SADT context diagram in figure 1 shows the inputs, outputs, controls, and mechanisms involved in domain analysis. Information is collected from existing systems in the form of source code, documentation, designs, user manuals, and test plans, together with domain knowledge and requirements for current and future systems. Domain experts and domain analysts extract relevant information and knowledge. They analyze and abstract it. With the support of a domain engineer, knowledge and abstractions are organized and encapsulated in the form of domain models, standards, and collections of reusable components. The process is guided by domain analysis methods and techniques as well as management procedures. The domain analysis process may be part of a software development environment.

This is an ongoing process of continuous refinement. As reusable resources are made available and new systems are constructed, they are used to refine existing domain models and to contribute to the reuse library. A formal domain language that isolates systems designers and builders from software construction details would be a prime objective of this refinement process.

The domain analyst plays a central role. He or she coordinates the whole analysis process. The domain expert and domain engineer play supporting roles in facilitating the input or acquisition phase and the output or encapsulation phase respectively. Typical domain models range

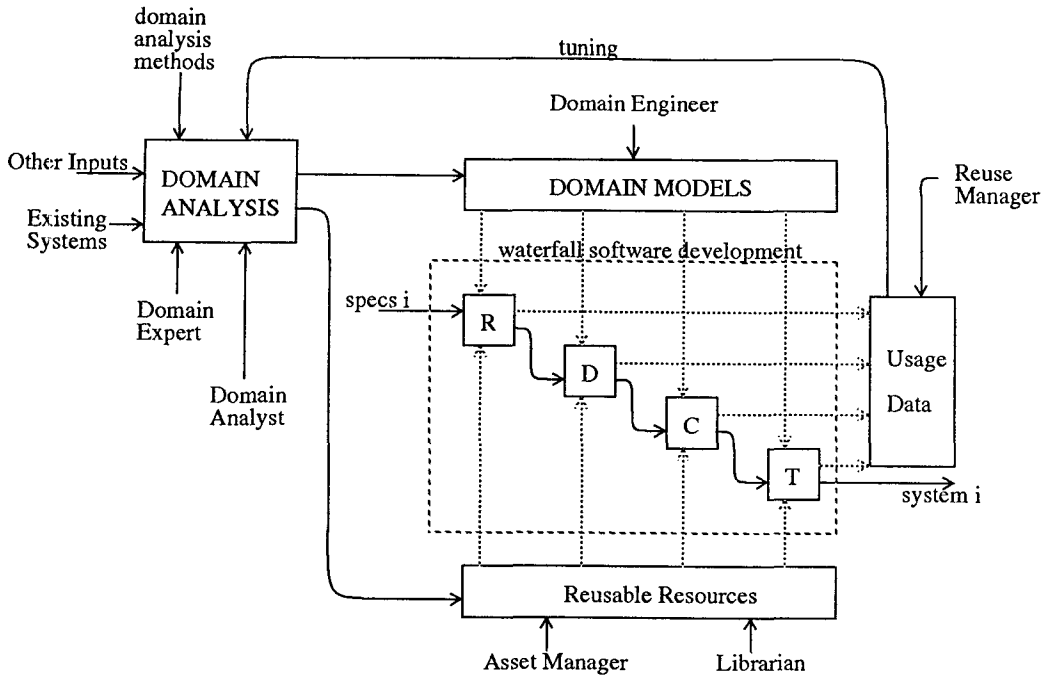

Figure 2. A Reuse Infrastructure

in level of complexity and expressive power from a simple domain taxonomy to functional models to domain languages. Standards may include requirements specifications and design methods, coding standards, development procedures (e.g., walk-throughs), management policies, and library maintenance procedures.

Reuse Infrastructure: Figure 2 illustrates one view of how domain analysis may be integrated into the software development process in the form of what we call the reuse infrastructure. Domain models, in a variety of forms, support (i.e., control) the different phases of software development. Reusable resources are selected and integrated in the new system. Reuse data is then collected and fedback to domain analysis for refining the models and for updating the library. As developed systems become existing systems they are also used to refine the reuse infrastructure. Although the waterfall model is used in this figure, other approaches like rapid prototyping could also be supported by a reuse infrastructure.

We have identified three specific roles critical to a reuse infrastructure: a librarian, an asset manager, and a reuse manager. All three complement the tasks of the domain analyst, expert, and engineer by managing the reusable resources and collecting information. The librarian task is to promote reuse by making assets available and easily accessible to potential reusers, the asset manager controls asset quality and standards compliance, and the reuse manager supports data collection relevant to domain analysis and coordinates the overall reuse effort.

A reuse library system is necessary to support all the tasks and processes of a reuse infrastructure.

6. Issues in Domain Analysis Research

Knowledge representation: How can we represent knowledge in a way that it is easily understood by humans and also machine processable? This topic has been the subject of extensive study in artificial intelligence and database modeling [CF82, BMS84]. Widely used approaches include E-R diagrams, predicate logic, semantic nets, production rules, and frames. Each approach offers its own unique features and trade-offs. Semantic net models, for example, offer explanatory power and rich conceptual associations but are difficult to implement and to maintain. Rule-based models do not offer the expressive power of semantic nets but are less difficult to implement and maintain.

Database systems also offer some answers to the knowledge representation problem. Although more limited in supporting artificial intelligence functions like inference and learning, database systems offer standard, widely-used technology that is efficient, easy to use, and easy to maintain.

Graphical models, like hypertext, offer an alternative to represent knowledge for ease of human understanding. In fact, hypertext has opened up new possibilities to use the computer as a communication and thinking tool. Through browsing, an analyst can discover objects and relationships in a domain. Hypertext technology has captured the attention of researchers and industry and several implementations are now available [Con87]. Visual programming is also an alternative well suited to represent and facilitate understanding of certain software workproducts like execution patterns, specification and design animations, testing plans, and systems simulation. Domain analysis research should benefit from the several ongoing research projects in visual programming [COM85].

There is a need in domain analysis research to integrate results from research efforts and emerging technologies in knowledge representation. What knowledge representation approaches best support the various activities and products of domain analysis? What techniques may be better suited to represent domain models, reuse libraries, and software products?

Knowledge Acquisition: What combination of knowledge representation and knowledge acquisition techniques is best suited for the different kinds of information required in domain analysis? There is a significant trade-off between knowledge representation and difficulty in knowledge acquisition. Populating a relational database, for example, is a much simpler task than creating a semantic net. Knowledge representation models that offer more explanatory power demand higher up-front investment and cognitive effort. There is a need to explore the best technique or combination of techniques that allow for better systematic reuse of software products.

Machine learning research has focused on approaches for automatic knowledge acquisition [MCM83, MCM86] while expert systems development has concentrated more on empirical techniques for extracting expert knowledge [COM86, HWL83]. Domain analysis research can benefit from both. The latter provides applicable practical guidelines in an attempt to formalize the acquisition process while the former demonstrates formal concepts and learning theories. Meta-knowledge, the procedural knowledge on how to use declarative knowledge, is crucial to domain analysis. It is widely accepted in the expert systems community that meta-knowledge is the hardest kind of knowledge to elicit from experts. Furthermore, meta-knowledge acquisition is a highly interactive process. The COMPASS project at GTE Laboratories, for example, spent several months interviewing digital switch maintenance experts to learn and extract key diagnostic heuristics. The experts analyze computer printouts on current switch performance and fault reports and are able to detect and predict malfunctions from observing traffic patterns and bottlenecks. Eliciting, capturing, and encapsulating such knowledge became the major undertaking for the project [Pre89].

Evolution of information: A key issue in a software development model based on reuse is the feed-back mechanism required to refine the process. There is a need for new software development models that integrate the notion of evolution and refinement into the process. A step in that direction has recently been taken by Basili et.al. from the University of Maryland [BR88]. Basili's group proposes a reuse-enabling software evolution environment model that explicitly models learning, reuse, and feedback activities. The integrating element in this model is an "experience base" that records software development experience and promotes "tailoring" and "generalizing" cycles on three levels of information: project specific, domain specific, and general or domain independent. An experimental prototype environment, TAME (Tailoring A Measurement Environment), is currently undergoing evaluation. The issues in maintaining an experience base, however, are similar to the issues of maintaining knowledge bases in expert systems. Domain analysis research should also benefit from models used in artificial intelligence for knowledge maintenance.

Validation: Complementary to feed-back and refinement is information validation. There is a need to know if the information used for domain analysis actually contribute to a better performance of the software development process. Furthermore, there is a need to know when a certain level of refinement has been achieved. An acceptable level could be, for example, when a design model or an architecture has been standardized and used systematically in developing new systems. However, for other components of the reuse infrastructure like the reuse library, validation of new components may be an ongoing process.

Research issues in the area of software processes [ICS87] deal with similar problems. As information changes there is a need to validate such changes. New paradigms that include these mechanisms are needed.

Methodological issues: To be practical and effective, domain analysis should be accessible and systematic. Domain analysts should be able to use standard domain analysis methods and techniques just as systems analysts and systems designers use currently accepted methods such as SADT, DeMarco's dataflow diagrams, and Yourdon's structured design. There is a need to develop a domain analysis methodology and to validate it with experiments and case studies. An effort in this direction is currently underway at the Software Engineering Institute, Pittsburgh. A research objective for the Software Reuse Group is to develop a domain analysis methodology by analyzing and consolidating the best features of current domain analysis practice. Although an initial step has been taken, the magnitude and interdisciplinary nature of the problem calls for a broader effort. There are methods, techniques, and tools from several other disciplines already available that when properly combined or adapted could offer significant support for domain analysis. Channeling of these efforts is a main task of domain analysis research.

6. Summary

The objective of this paper has been to provide a brief introduction to the area of domain analysis as seen from the software engineering perspective. Motivation and justification for domain analysis was presented in the first section. In the background section we addressed some of the research efforts in software engineering that are explicitly called domain analysis. In section 3 we analyzed characteristics of selected research efforts that are related to domain analysis.

The three basic definitions in section 4 were intended to provide a general framework not to bound this

emerging discipline to a specific terminology. We hope that by explaining with examples and relationships the reader will understand the concepts and that formal definitions will emerge as we learn more about domain analysis.

A model for the domain analysis process was also proposed. The concept of a library based domain infrastructure was introduced as an attempt to show how domain analysis is integrated into the software development process.

The last section gives a perspective on some of the research issues facing domain analysis. The nature of the process calls for a variety of multidiciplinary issues ranging from knowledge acquisition and representation to management and methodologies to cultural and social questions.

7. References

[AM88] W. Agresty and F. McGarry. *The Minnowbrook Workshop on Software Reuse: A Summary Report.* Contract NAS 5-31500, Computer Sciences Corporation, Systems Sciences Division, 4600 Powder Mill Rd., Beltsville, MD 20705, March, 1988.

[Ara88] G. Arango. *Domain Engineering for Software Reuse.* Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1988.

[Bar85] D. Barstow. Domain-specific Automatic Programming. *IEEE Transactions on Software Engineering*, **SE-11**(11)1321-1336, November, 1985.

[BMS84] M. Brodie, J. Myopolus, and J. Schmidt. *On Conceptual Modelling.* Springer Verlag, New York, 1984.

[Boo86] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, **SE-12**(2)211-221, February, 1986.

[BR88] V.R. Basili and H.D. Rombach. *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment.* **UMIACS-TR-88-92**, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, December, 1988.

[bsh88] Bass Harbor Workshop on Tools and Environments for Reuse, Bass Harbor, Maine, June 21-24, 1988. Participants proceedings.

[Cam87] *CAMP, Common Ada Missile Packages, Final Technical Report, Vols. 1, 2, and 3.* **AD-B-102 654, 655, 656.** Air Force Armament Laboratory, AFATL/FXG, Elgin AFB, FL, 1987.

[CF82] P. Cohen and E. Feigenbaum. *The Handbook of Artificial Intelligence, Vol. III.* W. Kaufmann, Los Altos, CA, 1982.

[Cle88] J. Cleaveland. Building Application Generators. *IEEE Software*, **5**(6):25:33, July 1988.

[COM85] *IEEE COMPUTER*, Special Edition on Visual Programming, **18**(8), August, 1985.

[COM86] *IEEE COMPUTER*, Special Edition on Expert Systems in Engineering, **19**(7), July, 1986.

[Con87] J. Conkin. Hypertext: An Introduction and Survey, *IEEE COMPUTER*, **20**(9)17-41, September, 1987.

[FN88] S. Fickas and P. Nagarajan. Critiquing Software Specifications. *IEEE Software*, **5**(6)37-47, November, 1988.

[HWL83] F. Hayes-Roth, D. Waterman, and D. Lenat (Eds.) *Building Expert Systems.* Addison-Wesley, Reading, MA, 1983.

[ICS87] *Proceedings of the 9th International Conference on Software Engineering.* Monterey, CA, IEEE Computer Society Press, March, 1987.

[LP79] R. Lanergan and B. Poynton. Reusable Code- The Application Development Technique of the Future, *Proceedings of the IBM GUIDE/SHARE Application Symposium*, pp: 127-136, October, 1979.

[Mat81] Y. Matsumoto. SWB System: A Software Factory. In *Software Engineering Environments*, H. Hunke (Ed), pp: 305-317, North-Holland, GMD, 1981.

[Mat87] Y. Matsumoto. A Software Factory: An Overall Approach to Software Production. In *IEEE Tutorial in Software Reusability*, Peter Freeman (Ed.), Computer Society Press, Los Alamitos, CA, 1987.

[MCM83] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, (Eds.) *Machine Learning, Vol. I*, Morgan Kaufmann, Los Altos, CA, 1983.

[MCM86] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, Editors. *Machine Learning, Vol. II*, Morgan Kaufmann, Los Altos, CA, 1986

[McC85] R. McCain. Reusable Software Component Construction: A Product-Oriented Paradigm. In *Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, CA, pp:125-135, October 21-23, 1985.

[Neig81] J. Neighbors. *Software Construction Using Components.* Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1981.

[Neig84] J. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, **SE-10**:564-573, September 1984.

[oom88] Domain Analysis: Object-Oriented Methodologies and Techniques- Workshop. OOPSLA'88, San Diego, CA, September 27, 1988. Participants proceedings.

[Pre89] D.S. Prerau. *Developing and Managing Expert Systems: Proven Techniques for Business and Industry.* Addison-Wesley, Reading, MA, 1989.

[Pri87] R. Prieto-Díaz. Domain Analysis for Reusability. In *Proceedings of COMPSAC'87*, Tokyo, Japan, (23-29), October, 1987.

[rip89] Reuse in Practice Workshop, Software Engineering Institute, Pittsburgh, PA, July 11-13 1989. Participants proceedings.

[rmi87] *Proceedings of the Workshop on Software Reuse.* Rocky Mountain Institute of Software Engineering, Boulder, CO, October 14-16, 1987.

[RW88] C. Rich and R.C. Waters. The Programmer's Apprentice: A Research Overview. *IEEE Computer*, **21**(11)10-25, November, 1988.

[TM84] D. Tajima and T. Matsumara. Inside the Japanese Software Industry. *IEEE Computer*, **17**(3)34-43, March, 1984.

[TC86] T.F. Thompson and W.J. Clancey. A Qualitative Modeling Shell for Process Diagnosis. *IEEE Software*, **3**(2)6-15, March, 1986.

[Vic60] B.C. Vickery. *Faceted Classification: A Guide to Construction and Use of Special Schemes.* Aslib, 3 Belgrave Square, London, 1960.