6

Domains, patterns, reuse, and the software process

Alfs T. Berztiss
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260, USA
(e-mail: alpha@cs.pitt.edu; fax: +412-624-8854)

and

SYSLAB, University of Stockholm, Sweden

Abstract

The business world of today is characterized by very rapidly changing business conditions that require rapid responses. The responses can be rapid only if we understand how to use models of the domain in which the business operates. Unfortunately there is considerable confusion regarding domain analysis, which seems to be due to the complex structure of a realistic domain model. In an attempt to remove some of the confusion, we establish a framework for defining this structure, and propose a software development process based on the structure. Our main contribution is the identification of three aspects of domain analysis, namely domain partitioning, refinement of domain models, and analysis of situations. Our emphasis is on how domain analysis can help software reuse.

Keywords

Business software system, domain analysis, domain knowledge, domain model, pattern, reuse, software process

1 INTRODUCTION

Domain analysis has several purposes. First, domain analysis is to allow all activities within a domain to be understood so that they can be improved. Domain analysis is thus an important component of business reengineering. Second, the conventions of specific application domains have to be well understood so that software systems deployed in these domains comply with the conventions. Third, a domain has to be described in such a way that reuse is enhanced. In particular, maintenance, which can be regarded as a kind of reuse (Basili, 1990), is to become more effective, i.e., is to allow rapid and effective changes to business software systems in response to changes in the domain. We speak of *business software systems* rather than *information systems* to emphasize that the software systems of business enterprises of today perform many control functions, some of which can be very intricate, e.g., programmed trading in stock markets.

Although the importance of domain analysis is now realized, software developers are still uncertain regarding the exact nature of domain analysis. Taking a very broad definition of "system", in most areas of system development, such as the design of roads, buildings, or manufacturing facilities, it is understood that there are general principles that define a discipline or an occupation, but that these principles have to be tailored to particular applications - an opera house and a hospital look and function differently. Domain analysis accounts for the differences. With regard to software, there is confusion as to the exact meaning of domain analysis in this particular context. Bruce Blum (1989) attributes the confusion to a lack of distinction between knowledge that relates to application domains, i.e., about problems to be solved, and knowledge that relates to implementation domains, i.e., about the tools, representations, and methods that are to help solve the problems. This lack of distinction is understandable because to computer scientists the implementation domain is also an application domain. Indeed, in most cases, it is the only application domain with which they are familiar.

Confusion is reduced by well-defined terminology. In what follows, we make a clear distinction between domain knowledge and domain models. *Domain knowledge* is the entire corpus of data, rules, and processes that characterizes a discipline. Some of it is codified in standards and handbooks, but most of it is distributed in the collective memory of practitioners of the discipline. Lately some domain knowledge has been expressed as patterns - see, e.g., (Coplien and Schmidt, 1995). A *domain model* is an abstraction that consists of only those parts of the domain knowledge that are relevant to a particular application. The purpose of *domain analysis* is to develop domain models.

Although at some point a domain model has to be linked to an implementation, here we shall consider applications without reference to implementation. We shall see that even with this restriction a domain model can have a very complex structure, including knowledge that extends to fairly detailed design decisions. One of our tasks is to establish a framework for defining this structure. A second task is to propose a software development process adapted to this structure.

Since software, including information bases, has become the most important asset of many organizations, our approach has to be broad. Thus, we have to realize that the main problem facing any organization today is rapid technological change, and that this change requires very rapid reactions to allow the organization to continue at the same level of operation, or even to survive. As an indicator of the rapidity of the change, in just five years between 1987 and 1992, 143 companies were replaced on the *Fortune 500*, a list of the 500 largest companies in the United States (Burrus, 1993, p. 169). We have discussed the change phenomenon as such elsewhere (Berztiss, 1995). Here we shall confine ourselves to application domain models, in particular models that are to allow rapid reengineering of a business system.

We shall consider application domain models along three dimensions. First, at every stage of development, ranging from requirements gathering to implementation, a business software system should be regarded as a modularized assemblage of components of three kinds - components that define the structure of an information base, components that indicate when and how the information base undergoes changes, and components that define processes. We discuss modularization and the three-way classification in Section 2. The second dimension relates to a refinement of an application domain model, and to how enterprises fit into domain models. This dimension is considered in Section 3. The third dimension, which we consider in Section 4, relates to situations. Similar situations can arise with different applications and in different enterprises, and domain analysis is to help arrive at common approaches to such situations. In Section 5 we introduce a composite software process for business systems in which we have incorporated our findings. Section 6 indicates what further work needs to be done.

2 DIMENSION I: COMPONENTS OF A BUSINESS SOFTWARE SYSTEM

In his survey of business software systems, Kroenke (1992) distinguishes between systems for transaction processing, management information, decision support, office automation, and executive support. This classification can no longer be regarded as current because in most modern business software systems these capabilities are intermixed. For example, an order fulfillment system could, in addition to the processing of regular orders, generate periodic summaries for management, use the summaries to assist in decisions of when to increase inventory by initiating manufacturing runs, replace explicit orders as the basis for initiating shipments by automatic monitoring of client inventories and produce surveys of long-term trends for top management. However, all these activities can be defined in terms of three basic components: a structured information base, procedures for changing the information base, which we call events, and processes. The first two components were considered the more important in the past, but processes are now being given special emphasis, particularly by advocates of business reengineering (Davenport, 1993; Hammer and Champy, 1993; Johansson *et al*, 1993).

It is often stated that in the early phases of software development one should merely determine *what* is to be done, and not *how* this is to be accomplished. It has been shown that such a separation of concerns is not feasible (Swartout and Balzer, 1982). This becomes most apparent when we consider how domain analysis relates to processes. We define a process as an ordered collection of tasks that is to achieve an objective. Domain analysis is to identify the tasks and how they they are linked, and this is "how" rather than "what" - in most instances only a domain expert has the knowledge of how the "what" is to be converted into a "how".

Further, the structure of the information base of an application is to be determined by a consideration of the object classes that the application deals with. In a banking application the classes could be the bank employees, customers, mortgages, accounts, etc. A software module can be constructed around each class. The identification of appropriate object classes should start off domain analysis. First, the names of the classes provide an initial vocabulary for talking about a domain. Second, a domain model should be modularized, and the object classes provide a natural modularization mechanism.

The identification of software modules with object classes or data types is the basic philosophy underlying the design of our specification and prototyping language SF - see, e.g., (Berztiss, 1995, pp. 176-195) or (Berztiss, 1990). Because of ambiguity problems with natural language (Wing, 1988) domain models should be expressed in a formal language. Our own preference for this is SF, because it partitions an application into modules based on data types, and within each module supports a clear separation of the three basic components introduced above: an SF schema defines an information base consisting of a set and functions on this set; a set of events defines operations that change the information base; and a set of transactions allows events to be linked into processes. The definition of a process in terms of linked events is precisely what Parnas (1972) recommended many years ago - far from being outdated, the recommendation is actually gaining importance as more complex systems are being constructed.

3 DIMENSION II: DOMAIN MODELS AND THEIR REFINEMENT

When we say that a domain spans various enterprises, and that an enterprise may span several domains, we have in mind a very broad definition of *domain* - we tend to think of an entire industry, such as the chemical industry or automobile manufacturing. Actually it is more appropriate to talk of professions or occupations, such as chemical engineering, accounting, or management. But, although a domain model can cover an entire profession, for software development the more appropriate models cover enterprises, or processes such as order fulfillment, or procedures for dealing with particular situations, e.g., exceptions. We have to realize that a profession, an enterprise, a process, and a situation can all be represented by domain models. We shall distinguish between an *occupation model*, an *enterprise model*, a *process model*, and a *situation model*.

This classification is orthogonal to the partition of a domain model into its information base, events, and processes, as discussed in Section 2. Note that we are using *process* in two different senses. A *process model* relates to a complete business process, such as order fulfillment. The *process component* of a process model defines the structural composition of the business process, but, in addition to the structural composition, the model of the business process has also to refer to an information base and to how changes in the information base are effected.

An occupation model captures the knowledge that relates to the practices followed by members of an occupation in general as an explicit structure. For example, in (Berztiss, 1992) and (Berztiss, 1995, pp. 45-58) we have identified twelve general principles of engineering, and shown how they apply to software engineering. This is an operational definition in the sense that the definition of software engineering is a specialization of a list of the things that all engineers do. Normative definitions have also been proposed (Paulk *et al*, 1993; Haase *et al*, 1994). They enumerate the capabilities that are needed at different levels of effectiveness of software development and describe the practices to be followed at each level. An example illustrates the distinction between domain knowledge and a domain model. In the context of the software engineering domain, domain knowledge tells us that software inspections disclose a certain percentage of faults in a software product - see, e.g., (Weller, 1993); this knowledge makes us include software inspections into a domain model for the software development process, but the knowledge as such has no relevance for the domain model. On the other hand, some of the practices recommended to be followed in an inspection would become embedded in the process component of the domain model.

The motivation for setting up a domain model for the process of software development is to improve cost, schedule, and quality. These are objectives that relate to the general strategy of an enterprise. A very effective method for formulating such aims has been CSF (Rockart, 1979), short for Critical Success Factors. In general, the aim of CSF analysis is to develop processes that will assist a company to reach its strategic goals. The determination of the goals is a function of top management. Some examples of what a company may wish to achieve: increase market share; accumulate a cash reserve for overseas expansion; reduce payroll costs.

The general strategy statement is converted into a list of specific goals with time limits. Continuing with the first example: the market share is to be 50% within two years of now. From this goal derive critical success factors for software: maximal use of embedded software in all products, reduction of cost to customers by software reuse, reduction of customer inconvenience by software quality improvements. The critical factors suggest that in the software process special attention is to be given to identifying opportunities for software embedding. The development process for embedded software could include (i) a task that searches a software reuse library for software units that could be adapted for this system; (ii) a task that determines the reuse potential of all software units being developed for this system; (iii) emphasis on validation throughout. At this level, then, we have highly specific detail, but the detail still still does not go beyond the expression of objectives.

An enterprise domain model starts out with an objectives formulation, the objectives give rise to processes, and the processes are broken down into components that we call situations. One way of looking at domain analysis is to regard it as the refinement of objectives into a structured collection of situation models. The purpose of the refinement is to assist in the understanding of an application and to indicate opportunities for reuse. It has been observed (Maiden and Sutcliffe, 1993) that reuse requires considerable human participation. Just as we have distinguished between domain knowledge and domain models, so we should also distinguish between knowledge reuse and model reuse. The former is people-oriented, the latter provides opportunities for mechanization. We interpret domain refinement as a transition that starts at a level at which the domain model can only be interpreted by people because the knowledge incorporated in the model cannot be properly formalized, and reaches a level at which formalization is feasible. In other words, domain analysis extracts from and elaborates on those components of the enterprise model that can be described in a formal language.

The objectives of an enterprise are sometimes expressed in the form of *patterns*. A collection of patterns gathered by Coplien (1995) includes establishing the size of an organization, building teams by self-selection, establishing an apprenticeship program, and using scenarios to improve ultimate customer satisfaction. Whitenack's (1995) catalogue includes requirements gathering, consideration of customer expectations, establishing good relationships with customers, and, at a specialized level, several patterns that deal with the development of a system of domain objects. The key practices of the SEI Capability Maturity Model (Paulk *et al*, 1993a) can also be interpreted as patterns. Note that all these patterns relate to enterprises that develop software. The outlook has to be much broader.

Patterns can be interpreted as critical success factors. Guided by Alexander's (1979) work on patterns in architecture, the description of a pattern generally includes a statement of a problem to be solved, a discussion of the problem in which the context of the problem and the forces in effect are stated, and a solution. In CSF terms, the problem statement relates to general strategy, and the solution identifies critical success factors. The novel aspect is that patterns are meant to be reused.

Let us now consider a concrete example, a car rental company. We shall first identify a set of very general patterns, and then adapt them to car rental. The general patterns relate to reservations, actual rentals, acquisition and disposal of inventory, tracking of transaction trends, and management of distributed sites. The reservations and rental patterns arise in the hotel, airline, car rental, and formal wear rental businesses. This is by no means an exhaustive list - Maiden and Sutcliffe (1992) point out the similarity between a theatre reservation system and a university course registration system; we note that both are instances of the reservation pattern. Tracking of transaction trends should be practiced by every enterprise - a downward trend is a danger signal; an upward trend of car rentals should trigger inventory build-up; an upward trend in hotel occupancy rates can justify an increase in room charges. Management of distributed sites is particular to car rental: a car may be picked up and dropped off at different sites, and transaction trends at different sites may suggest a redistribution of cars.

The general patterns are next to be transformed into processes that are specific to the car rental business. Typically the reservation/rental process will have the following steps: (a) a customer reserves a car of a particular type for a certain period of time, but note that the "reservation" may be a walk-in event; (b) the customer arrives, presents a credit card, and gets the car; (c) the customer returns the car, and the car is inspected for damage, the fuel level in the tank, and the mileage for this rental; (d) the customer finalizes payment; (e) the car is prepared for the next customer. What complicates matters are exceptional conditions. We list a few: no cars are available; a car of the requested type is not available; customer decides to change the type after arrival; a customer does not show up, or the credit card check fails; a car is returned late, or is not returned at all; a car has been damaged to such an extent that an insurance claim has to be lodged.

The basic reservation pattern has several extensions. One relates to group reservations, when, for example, a block of hotel rooms is reserved for participants in a conference. Another is part of order fulfillment - when an order is received, and it cannot be filled in its entirety, those items that could be shipped are set aside, i.e., reserved, while the customer is asked whether a partial fulfillment of the order would be satisfactory. The realization that in these special instances we can still make use of the basic reservation pattern, and the knowledge that allows the extensions to be carried out is at this time beyond the capabilities of software systems. This is of little concern because people can be very efficient in adapting specifications. Thus, it took the author no more than 15 minutes to convert an SF specification of a software system for controlling the kind of high-tech baggage lockers one sees in some French railway stations into a specification of a software system for vending machines.

We now have to relate the four models we introduced at the beginning of this section to patterns. The occupation and enterprise models suggest what business software systems are needed and what patterns are relevant in the setting up of the systems. Specifically, the occupation model allows a pattern library to be partitioned according to occupational needs, which facilitates pattern retrieval. A process model imposes structure on a pattern, and a situation model relates a process to the setting in which it is to operate - in particular, exceptional conditions in the operation of a process are covered by the situation model.

4 DIMENSION III: SITUATIONS

In our example of car rental we defined a five-step normal car rental process, and indicated several exceptions. The entire process, including exceptions, can be defined in a formal specification language such as SF. However, we should avoid the construction of process definitions from scratch for each application. It appears to us that the biggest obstacles to effective reuse are a preoccupation with code reuse, which we find of little importance except in strict maintenance efforts, and the identification of software design as a specific separate step of the software process.

We interpret design to have two aspects. In mature engineering disciplines design allows a full evaluation of the properties of an artifact before the artifact is constructed, but for software design this is, for the most part, an ideal that is yet to be attained. The second aspect is that design converts something unstructured into something that has a well-defined structure. Within software engineering this is a process that starts with imposing modular structure on a natural-language requirements document, produces a specification expressed in terms of components that present well-defined interfaces to other components, and converts the specification components into code components. Since the last step can be fully automated, specification reuse rather than code reuse has the real importance.

Reuse of specifications has several attractive features. First, a specification in a language such as SF is generally easier to read and understand than code. Second, since in a specification the application-related content can be separated from the information needed to make the specification executable, a specification is easier to change than the corresponding code. Third, specification fragments, which should be self-contained components, can be related to patterns: analysis of a pattern suggests what generic specification components there should be. Retrieval is then not of the specification components directly, but of the patterns to which they belong, and this is easier to implement because the number of patterns should be much smaller than the number of components.

Analysis of situations is to indicate which patterns are relevant, which of the components "indexed" by a pattern are to be retrieved, and how they are to be modified. Returning to car rental, we identified five applicable general patterns, and saw how a combination of the reservation and rental patterns was to be converted into a specialized process. This specialization is one instance of situation analysis.

A more interesting aspect of situation analysis relates to exceptions. Here, too, we can identify general patterns - a shortage pattern covers the cases of no cars available and no cars of a specified type available; a transaction termination pattern covers no show and no credit; change of type of car is to be subsumed by a more general pattern that relates to any change in a reservation; damage to a car relates to two patterns, an insurance claim pattern and a repair pattern; no return of a car belongs to a loss pattern. Situation analysis requires much skill, skill that can be supplied only by people. For example, it has to be realized that damage or loss of a car affects accepted reservations, and that termination of a transaction releases a reserved car. Termination should suggest further that waiting lines be established, leading to the retrieval or creation of a waiting-line pattern.

5 A SOFTWARE PROCESS

Glass and Vessey (1995) emphasize the need to distinguish between the intrinsic nature of the problems that a software system is to solve, and the way in which they are to be solved. They propose that there be two domain taxonomies, one that characterizes applications, and another that characterizes solution methods, i.e., relates to the software development process, and that the same classification criteria are to be used for both taxonomies to facilitate mapping between the two. In our investigation of a process for domain modeling we shall be influenced by such a mapping, but will not be explicitly concerned with it.

A software development process based on domain analysis has two extreme versions - pure top-down development that starts with a set of patterns, and "transforms" the patterns into a formal specification of a software system in terms of reusable components, and a pure bottom-up approach in which by a high-level domain-specific objective is a goal to be attained by construction of a software system from existing components, which may have to be adapted to the needs of the application. The development of a business software system is likely to follow a model that lies between the two extremes. In (Berztiss and Bubenko, 1995) we introduced a software system development model, which, modified and extended, is shown here as Figure 1.

In this model, system development begins with objectives and ends with a business software system. The objectives determine a domain model that consists of an information base, events, and processes. This somewhat informal model is converted into a formal functional specification. There are also nonfunctional aspects of a software system, such as usability, which can rarely be stated in formal terms - they are represented by the "Nonfunctional requirements" box. This development process is defined in terms of various concepts, and different roles in the process are allocated to actors, which can be people or software tools. Several versions of the system can be generated, and, during the lifetime of a particular version, it is represented in different forms, such as requirements, design, and code. There must always be a clear understanding of which version a particular representation belongs to (version control), and that all representations of a version are consistent (configuration control). The more enclosed a box, the more specific is its role in the process. The box for version and configuration control, and the three boxes for the libraries are not enclosed, which means that they are not influenced by what particular software product is being developed, i.e., they are fully application independent.

At some stage the process makes use of patterns from a basic pattern library and an exception pattern library, and these libraries in turn reference a component library. The corresponding boxes in Figure 1, which represent the major extension of the earlier process model, show that these libraries provide input to the process, but it is left open when and how this happens.

Under top-down development, the definition of objectives takes place without reference to the patterns, and so does the setting up of the three-component structure. Only when the next transition is made, to functional specification, are the pattern libraries consulted, and this leads to the introduction of existing components into the functional specification. Under bottom-up, the pattern libraries drive development. Reference to these libraries suggests what objectives are to be formulated, i.e., a functional specification may be created first, and an objectives formulation added as an afterthought. Whatever the approach, the component library is encapsulated in the sense that access to the components is only via the patterns to which the components are keyed. This is to prevent copying of components without proper understanding of how appropriate they are for a given application. Poor understanding of application domains is a major problem in software development (Curtis *et al*, 1988). Encapsulation is to make developers acquire better understanding.

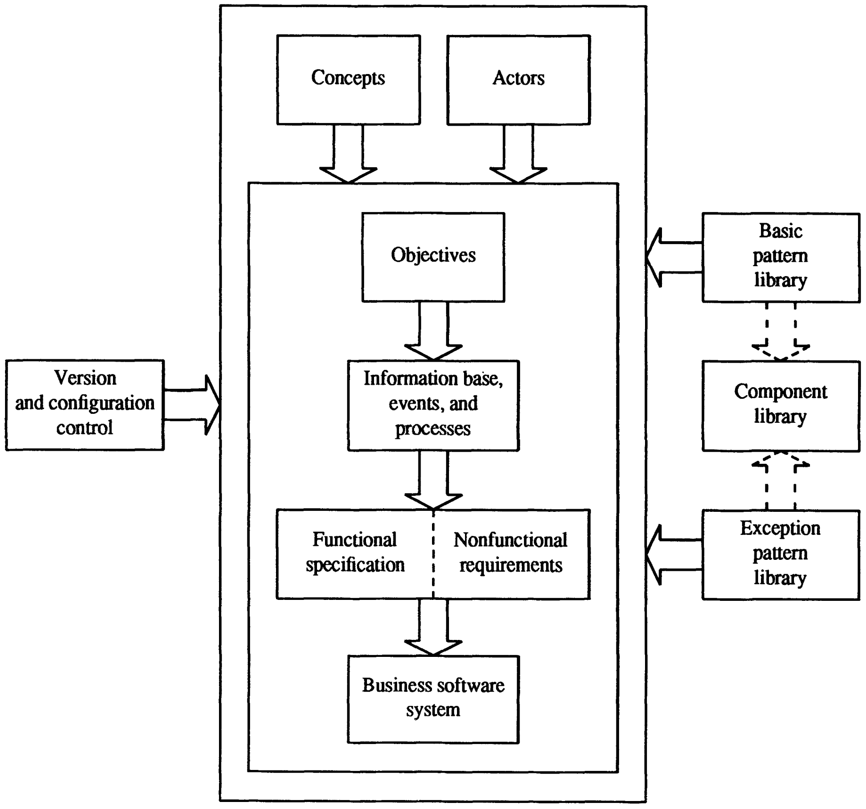
Concepts
Actors
Objectives
Information base, events, and processes
Functional specification
Nonfunctional requirements
Business software system
Version and configuration control
Basic pattern library
Component library
Exception pattern library

Figure 1 A reuse-based software development model

6 AN AGENDA FOR THE FUTURE

In Section 3 we introduced a refinement scheme that transforms an informal upper-level enterprise model into a formal situation model. The latter can be interpreted by software. So can a process model because it is no more than an assembly of situation models, but enterprise and occupation models need to be interpreted by people. A major research project undertaken by Lenat (1995) aims at formalizing what is loosely called "common sense". In our context we should determine those aspects of "common sense" that relate to enterprise and occupation models, and try to arrive at definitions of such aspects in formal terms. To some extent this has already been done - large parts of enterprise and occupation models can be expressed as ***business rules***, and business rules have been coupled to representations of processes in terms of data flow diagrams (McBrien and Selveit, 1995), but we need to find a way of setting up semi-formal domain models made up of business rules and informal knowledge components, and a methodology has to be developed for using such semi-formal models effectively in the development of application software.

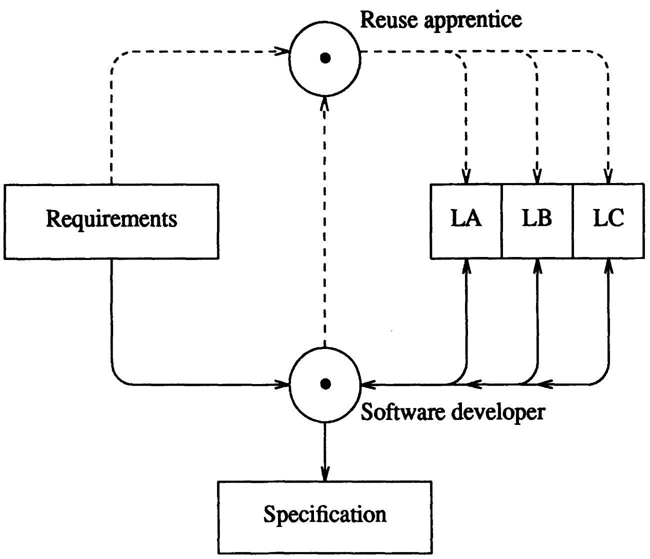

Figure 2 The development of a specification

As a preliminary to this, we should clarify how exactly the libraries of Figure 1 are to provide components of a formal functional specification. Figure 2 gives an indication of how this can be done. The three libraries are represented by boxes LA, LB, and LC. The box labeled "Requirements" corresponds to the "Information base, events, and processes" box of Figure 1. Software developers are to compose specifications by consulting the libraries. In this they are to be helped by a tool. This tool is labeled "reuse apprentice", and is to be an adaptation of the software apprentices described by Rich and Waters (1990). A software apprentice is a tool that learns how to match requirements with components of the libraries. In our context, in the first instance, the apprentice would help a software developer navigate through the libraries. Some hints on how to approach the design of the apprentice can be found in (Marques *et al*, 1992).

We referred earlier to collections of patterns for the software process itself. We need to set up general pattern libraries and establish a retrieval scheme based on the needs of various occupations, e.g., safety engineering or management of rental companies, and of enterprises. As regards the latter, the objectives of an enterprise can be linked to skills needed to achieve them, so that the patterns corresponding to specific objectives can be accessed via an occupation model. These relationships need to be elaborated. The taxonomies collected by Glass and Vessey (1995) provide a starting point.

In discussing patterns and components it was indicated that only the latter were to be defined in a formal language. Actually patterns can be formalized to some extent, as cliches or templates. We have used SF to define some safety-critical situations as formal process specifications in which only some application-specific parts were left undefined (Berztiss, 1996). These safety cliches are instances of exception patterns. Similarly, we have developed the general purpose cliches that were found relevant for the car-hire business. It remains to be established when it pays to formalize, i.e., whether it is easier to customize formal or informal general patterns.

Let us return to the match-up between application taxonomies and development taxonomies. The role of software architecture in this task should be investigated. Some examples of architectural styles: pipeline, data abstraction (object-oriented), implicit invocation (event-based), repository, and subroutine-based (Shaw, 1995). Our SF subsumes data abstraction and implicit invocation, and is adequate for the specification of business software systems. However, it is not suited for the specification of processes that operate on data streams, for which a pipeline architecture should be considered.

7 REFERENCES

Alexander, C. (1979) *The Timeless Way of Building*. Oxford University Press.

Basili, V.R. (1990) Viewing maintenance as reuse-oriented software development. *IEEE Software* **7**, (1), 19-25.

Berztiss, A. (1990) Formal specification methods and visualization. In *Principles of Visual Programming Systems* (ed. S.-K. Chang), 231-290, Prentice Hall.

Berztiss, A.T. (1992) Engineering principles and software engineering. In *Proc. 6th SEI Conference on Software Eng. Educ., 1992* (Springer-Verlag LNCS No.640), 437-451.

Berztiss, A.T. (1995) *Software methods for business reengineering*. Springer-Verlag.

Berztiss, A.T. (1996) Unforeseen hazard conditions and software cliches. *J. High Integrity Systems*, to appear.

Berztiss, A.T., and Bubenko, J.A. (1995) A software process model for business reengineering. In *Information Systems Development for Decentralized Organizations*, 184-200, Chapman & Hall.

Blum, B. (1989) A paradigm for the 1990s validated in the 1980s. In *Proc. AIAA Conf. 1989*, 502-511.

Burrus, D. (1993) *Technotrends - How to Use Technology to Go Beyond Your Competition* (with R. Gittines). Harper Business.

Coplien, J.O., and Schmidt, D.C, eds. (1995) *Pattern Languages of Program Design*. Addison-Wesley.

Coplien, J.O. (1995) A generative development-process pattern language. In Coplien and Schmidt (1995), 184-237.

Curtis, B., Krasner, H., and Iscoe, N. (1988) A field study of the software design process for large systems. *Comm. ACM* **31**, 1268-1287.

Davenport, T.H. (1993) *Process innovation: reengineering work through information technology.* Harvard Business School Press.

Glass, R.L., and Vessey, I. (1995) Contemporary application-domain taxonomies. *IEEE Software* **12**, (4), 63-76.

Haase, V., Messnarz, R., Koch, G., Kugler, H.J., and Decrinis, P. (1994) Bootstrap: fine-tuning process assessment. *IEEE Software* **11**, (4), 25-35.

Hammer, M., and Champy, J. (1993) *Reengineering the corporation: a manifesto for business revolution.* Harper Business.

Johansson, H.J., McHugh, P., Pendlebury, A.J., and Wheeler, W.A. (1993) *Business Process Reengineering: Breakpoint Strategies for Reengineering.* Wiley.

Kroenke, D. (1992) *Management Information Systems.* McGraw-Hill.

Lenat, D.B. (1995) CYC: a large-scale investment in knowledge infrastructure. *Comm. ACM* **38**, (11), 33-38.

Maiden, N.A., and Sutcliffe, A.G. (1992) Exploiting reusable specifications through analogy. *Comm. ACM* **35**, (4), 55-64.

Maiden, N.A.M., and Sutcliffe, A.G. (1993) People-oriented software reuse: the very thought. In *Advances in Software Reuse* (eds. R. Prieto-Diaz and W.B. Frakes), 176-185, IEEE Computer Society Press.

Marques, D., Dallemagne, G., Klinker, G., McDermott, J., and Tung, D. (1992) Easy programming: empowering people to build their own applications. *IEEE Expert* **7**, (3), 16-29.

McBrien, P., and Selveit, A.H. (1995) Coupling process models and business rules. In *Information Systems Development for Decentralized Organizations*, 201-217, Chapman & Hall.

Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15**, 1053-1058.

Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, C.V. (1993) Capability Maturity Model, Version 1.1. *IEEE Software,* **10** (4), 18-27.

Paulk, M.C., Weber, C., Garcia, S., Chrissis, M.B., and Bush, M. (1993a) Key practices of the Capability Maturity Model Version 1.1. SEI Report CMU/SEI-93-TR-25, Software Engineering Institute of Carnegie-Mellon University.

Rich, C., and Waters, R.C. (1990) *The Programmer's Apprentice.* ACM Press, 1990.

Rockart, J.F. (1979) Chief executives define their own data needs. *Harvard Business Review* **57**, (2), 81-93.

Shaw, M. (1995) Architectural issues in software reuse: it's not just functionality, it"s the packaging. In *Proc. ACM SIGSOFT Symp. Software Reusability* (special issue of *ACM SIGSOFT Software Engineering Notes*), 3-6.

Swartout, W., and Balzer, R. (1982) On the inevitable intertwining of specification and implementation. *Comm. ACM* **25**, 438-440.

Weller, E.F. (1993) Lessons from three years of inspection data. *IEEE Software* **10**, (5), 38-45.

Whitenack, B. (1995) RAPPeL: a requirements-analysis--process pattern language for object-oriented development. In Coplien and Schmidt (1995), 259-291.

Wing, J.M. (1988) A study of 12 specifications of the library problem. *IEEE Software* **5**, (4), 66-76.