Creating Domain Specific Libraries: a methodology, design guidelines, and an implementation

Marcel Becker*

Robotics Institute
Carnegie Mellon University
Pittsburgh, PA. 15213

Jorge L. Díaz-Herrera†

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA. 15213-3890

Abstract

In this paper we present an approach for building libraries of reusable software components that addresses the tension between design-with-reuse and design-for-reuse. The approach is based on a hierarchical model that assumes four levels of reusability. The design guidelines for developing components according to this methodology are summarized and an application demonstrating how the products of a domain analysis technique can be mapped into this hierarchical model is described.

The application is a reactive scheduling architecture for manufacturing operations, and the domain analysis technique selected is the Feature Oriented Domain Analysis[11] developed by the Software Engineering Institute at Carnegie Mellon University. The main objective of the paper is to establish the connection between the domain analysis products and the different reusable levels identified by the proposed methodology.

1 Introduction

The software reuse process includes activities related to the identification of the desired components for a specific application, the detection of the availability of such components, and the analysis of the necessary adaptation to incorporate the components into the new system under development. From a pragmatic point of view, the basic thrust of reuse is the capability to integrate a coherent working system of interconnected software components, a process known as Design-with-Reuse, or *DwR* for short. This presupposes the existence of component libraries for which components are acquired and incorporated in a process known as Design-for-Reuse, or *DfR* for short. These two complementary reuse processes have conflicting goals[8]. From a DfR perspective, the components should be created aiming at providing similar functionality to a number of different but related applications. That is, the objective of DfR is to provide a flexible component. The emphasis here is on creativity.

From a DwR perspective, the needed components, providing a very specific functionality, should be selected from a stock of reusable software components. The objective of DwR is to save time and effort by retrieving an existing component rather than developing it anew. The emphasis is on understanding.

Seen under this perspective, DfR and DwR conflict in at least the following two ways.

1. The DfR view of a library is one of many heavily parameterized components, whereas the DwR view of a library is one of a few ready-to-use components.
2. When developing for reuse, the designer knows the component, but not the context in which it will be used; the need for flexibility breeds complexity. In contrast, when developing with reuse, the designer knows the context of usage, but not the component; the need for understanding demands simplicity.

Independent of the perspective considered, the identification of the needed components requires a special analysis of the application domain. By *domain,* in this context, we mean the set of systems or applications that share some functionality. The identification of commonalities across similar software systems as well as the dimensions along which the systems differ is one of the preconditions to achieve successful software reusability. An analysis process called *domain analysis* is one of the techniques that can be applied to meet this requirement[11].

In this paper, we concern ourselves with the use of domain analysis techniques and the application of component design guidelines to address this conflict directly. Although there are other aspects associated with the reuse process, as illustrated in Figure 1, we deal here only with issues related to source-code components.

*This work is sponsored by the Robotics Institute, Carnegie Mellon University and Conselho Nacional de Pesquisa e Desenvolvimento Tecnológico, Brasília, Brazil.

†This work is sponsored by the U.S. Department of Defense.

Figure 1: (a) The two main phases of software reuse: Design-for-Reuse (DfR) and Design-with-Reuse (DwR). (b) Conflicting goals between DfR and DwR.

2 Differentiating Components

Resolving the conflicting goals of DfR and DwR requires several steps, each one of them giving rise to different component levels. This approach works by isolating dimensions of the software not strictly necessary to the problem's requirements, and highlighting the fundamental role of strict layering in the organization of component libraries.

We propose a library organization based on a layered collection of components. Figure 2 summarizes the proposed library organizational model. Two fundamental component layers are identified, each partitioned into two levels, thus yielding a library structure with four levels. This simplifies both complex systems interactions and software construction, and achieves the effect of differentiating components, from the more general, simple, and smaller to the more specific, sophisticated, and bigger. These layers explicitly exploit design techniques known to be useful for DfR from those useful for DwR: the two top layers, grouped under the heading *Outer Core,* support DwR; and the two bottom layers, grouped under the heading *Inner Core,* emphasize DfR. The bridge between the Inner and Outer core is established by the two intermediate layers. Each level has different reuse goals, and each requires different techniques.

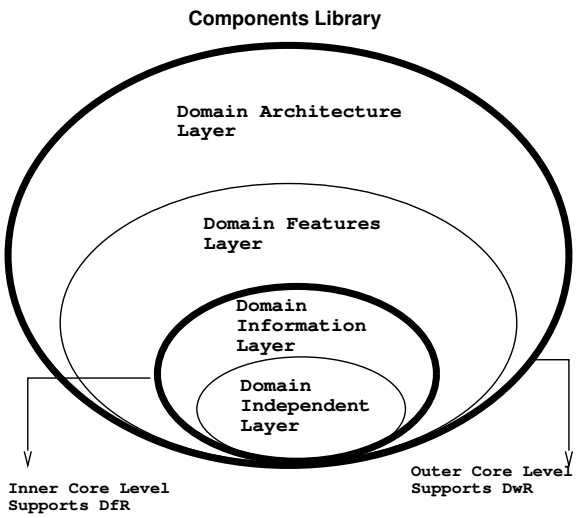

Figure 2: The Reuse Levels. The lower layer (inner core) emphasizes Design-for-Reuse: it builds up the functionality that will be reused. The upper layer (outer core) emphasizes Design-with-Reuse: it narrows the scope of the components down to the context of the reusing application. Intermediate layers bridge the gap between DfR and DwR.

2.1 The Inner Core

The inner core is driven primarily by DfR considerations, supported basically by abstract data types and abstract state machines (ADT/ASM) concepts. Components at this level typically represent bindings to domain-independent components, allowing interaction and interoperability among all the components. The inner core also collects the most basic domain-specific components (typically a semantic net of interrelated data structures). The inner core is further divided into two layers or levels of components as follows:

Layer 1: Domain-Independent (Bottom or Innermost) Components

This level of reuse has been known for a long time in the form of fundamental structures and algorithms. More recent is the emphasis on making fundamental algorithms and data structures available under the form of software components (e.g., Booch Components [4], GRACE [5]).

In this level it is also convenient to include even more complex components, such as language bindings for standard global domain concepts. The difficulty with the more complex components comes from the variety of nuances people read into the concepts represented by the components.

The result is a combinatorial explosion of functionality. For example, user interface management components offer data types by the dozen, windows, icons, widgets, buttons, scroll bars, etc., and literally hundreds of operations.

So instead of considering the domain overall as the basis for reuse, we take each one of the data types as a component in its own right. Then the design issues become similar to those for algorithms and data structures.

Layer 2: Domain Information (Intermediate) Components

In this layer, the components still represent fairly elementary concepts, but they are linked to specific contexts, i.e., domain concepts. They typically contain the description of the specific data structures that correspond to abstraction of domain entities and their inter-relations (in the form of an entity-relationship diagram or semantic net). These components serve as the bridge between the inner core and outer core. Domain analysis techniques (see below) detect the relevant domain concepts, their variances and uniformity.

The difficulty in this layer comes from the particular characteristics of a given domain. A collection of relatively small class hierarchies would help the designer more than a single large hierarchy.

2.2 The Outer Core

The outer core is driven by DwR considerations. The outer core represents the backbone of the software architecture, and thus must be capable of providing sockets into which we can plug domain-specific components. At this level, reuse is seen as more than just acquiring code; it consists of reusing designs. The components are collections of abstract components connected in ways that represent abstract, semi-finished domain designs and applications generated by concrete versions of these abstract designs. The components can be organized using techniques such as *frameworks* and *toolkits*[10].

Layer 3: Domain Features (Intermediate) Components

This type of reusable software is made of intermediate components that are neither a simple data structure nor a complete subsystem. Authors have coined many terms to describe means for reducing the gap between low-level, unspecific notions and high-level, specialized concepts, e.g., mediation and glue[14]. To conjure the image of a software factory, another nomenclature could be *semi-finished domain-specific components.* The artifacts at this level are ready for incorporation into an application, wherein they will materialize as finished components. The organization of these components can borrow, for example, from frameworks[10] or Domain Specific Software Architectures (DSSA) technology[15].

Typical activities involved in this level of design include grouping several abstract data types together, choosing a subset of an abstract data type's operations, providing common values for parameters, making inherited operations directly visible, making interfaces match, reconciling behaviors, etc. Such domain-specific software components make certain assumptions about the functionality the system will implement – they represent the features provided by the system for a particular application domain.

Layer 4: Domain Architecture (Top or Outtermost) Components

Finally, after these intermediate steps, the applications import the domain features components and make them usable by filling in the last parameters. Application developers and domain specialists negotiate the format of the semi-finished components. The domain experts delve into their toolbox represented by the basic features to build the buffering layer the applications will use.

Generation, such as generic instantiation, is the typical activity of this level. This is particularly supported by the notion of *toolkits* [10]. Usually toolkits are built on top of frameworks.

Design-with-reuse benefits from the dent the lower levels have made into the complexity of the components.

3 Creating a Domain Specific Reusable Library

This section summarizes the domain analysis performed in order to create a library of reusable components to be applied in the construction of a reactive scheduling system for production environments. The objective of the analysis process is to identify the required domain objects and functionalities and to map these requirements into the different component library layers identified in the previous section.

3.1 Feature Oriented Domain Analysis

Domain analysis identifies, collects, organizes, and represents most of the relevant information needed to design reusable components: it identifies the scope of the application, the objects of the domain, the different needed functionalities and how they vary across applications in a domain. The information is obtained from the study of the features and development history of existing applications, knowledge provided from domain experts, and the underlying theory.

The number of publications on domain analysis has greatly increased over the last ten years and several methodologies are currently available. A review of some domain analysis methods is presented in [11] and an extensive domain analysis bibliography can be found in [9]. Among the available methodologies, the *Feature Oriented Domain Analysis* [11, 7] developed by the Software Engineering Institute at Carnegie Mellon University, was a natural selection since the development group provided full support.

The feature-oriented concept is based on the emphasis this method places on finding the features or functionalities usually expected or desired in applications for a given domain. The analysis process, according to the selected methodology, is characterized by three basic phases, and each phase has specific procedures and products.

Figure 3 shows the three main phases of the Feature Oriented Domain Analysis methodology, the inputs needed at each phase, the final products, and how each phase maps into the design layers identified in section 2. More details about this mapping will be discussed in the next sections.

The validation of the domain analysis is discussed in [12]. The domain model products should represent the relevant information about the objects and functionality of a family of similar systems in a domain. Validation of the model is obtained by reproducing known applications through the selection of specific features and building of a prototype system. Variations between the prototype behavior and expected results indicate problems in the description of the model. The final validation is the system implementation. In the current analysis, the prototype has not been implemented yet but the products have been partially validated by presenting the products to experts in the field of reactive scheduling.

3.2 The Reactive Scheduling Problem

This subsection briefly introduces the reactive scheduling problem and the system architecture. Although a detailed description of the reactive scheduling domain is out of the scope of this paper, some background information is useful for the sake of clarity. The system architecture is based on a hierarchical reactive scheduling model proposed by Morton[16].

The objective of a scheduling system, in a general sense, is to assign *resources* to *operations* or *activities* (or vice-versa) in order to obtain a desired output over time. The type and characteristics of activities and resources vary from application to application. The schedule can be considered the representation of the state of the constraints imposed on the processing of the activities.

To make the concepts clear, consider the scenario of scheduling manufacturing operations in a factory. In this particular application, a sequence of operations has to be processed on a set of machines or work areas. The output of these operations is a set of *products* or *parts.* The objective of the production is to satisfy an external *demand* for some particular type and quantity of products. The demand is represented as an *order,* establishing a release date (the date the demand is available to the system), a due date, (the date the demand is expected to be satisfied), a product to be produced, and the quantity of the product. Each product defines a set of operations whose processing should follow a fixed sequence. Each operation requires a certain amount of *resource capacity* over time. The assignment of operations to resources should respect *capacity* and *precedence constraints* and be guided by a set of *preferences.*

Scheduling systems can be classified according to different dimensions. Two dimensions that are particularly useful for the present work are related to the schedule representation and the schedule generation strategy. According to the first dimension, the schedule generated can have different formats. It can range from representations that precisely specify the start and end times of each operation on each resource (interval-based schedule) to representations that assign only a relative importance to each operation (price-based schedule) and determine start and end times by simulating or actually executing the schedule. In the second case, operations are processed according to resource availability and an agent called a *dispatcher* is responsible for releasing operations to resources according to the sequence established by this priority list.

Concerning the schedule generation strategy, systems can range from pure generative off-line schedulers to pure reactive real time schedulers. Pure generative scheduling systems make decisions based on a static model of the system; that is, they usually work under the assumption that resources are always operational during their availability interval, and that nothing will go wrong during the actual execution of the schedule. On the other extreme, pure reactive scheduling system makes decisions on real time: at each decision point, the current state of the system guides the scheduling process.

To obtain a coordinated behavior, practical production environments usually require some kind of advanced plan. At the same time, however, these environments are subject to a number of disruptions like machine breakdowns and operations delays. These disruptions or uncertainties will invalidate the advanced plan. Therefore, the solution of the scheduling problem must provide some kind of mechanism to generate schedules that account for these disruptions in advance or that repair the schedule as disruptions occur. Solutions that incorporate generative and reactive components in the same system are common. In these systems, a basic schedule is generated by an off-line component. When the schedule is executed, if conflicts are identified, a repair action is generated.

The architecture to be implemented using the reusable components assumes the existence of a generative component external to the system, and an internal reactive component: the initial sequence of activities and resource assignments is established by an off-line external scheduler; once this schedule is generated, a priority or price is computed for each operation. The priority list is then used by a real-time dispatcher to send operations to be executed. As disruptions occur, an analysis process is triggered. According to the output of the analysis, a certain kind of reaction is performed to repair the schedule. The detailed description of the architecture can be found in [2, 3].

4 Context Analysis

The first phase of the domain analysis methodology is the Context Analysis. The purpose of this phase is to define the scope of the domain. This phase identifies the sources of input, the desired output, and the data

Phase	*Inputs*	*Activities*	*Products*	*Layer*
Context Analysis	**Operating Environment Standards**	**Context Analysis**	**Context Model**	**Domain Independent Layer**
Domain Modeling	**Application Domain Knowledge**	**Information Modeling**	**Information Model**	**Domain Information Layer**
	Features Context Model	**Features Analysis**	**Features Model**	**Domain Features Layer**
	Domain Technology Context Model Features Model Information Model Requirements	**Functional Analysis**	**Functional Model**	**Domain Features Layer**
			Behavioral Model	
Architectural Modeling	**Implementation Technology, Context Model, Features Model, Information Model, Design Information**	**Architectural Modeling**	**Structured Executive**	**Domain Architecture Layer**
			Subsystem Model	

Figure 3: Domain Analysis – Feature Oriented Domain Analysis Methodology

storage requirements. Once the scope is defined, the relations between the external and internal elements are analyzed and the variability of these relations are evaluated.

Becker in [2] analyzed some knowledge-based reactive systems and Smith in [18] made a much more complete survey of knowledge-based production systems. Armed with this information, we were able to identify commonalities and differences in the systems and it was also possible to establish the scope of the system and the interactions between a reactive scheduling system and its external environment.

The Context Model is composed of a *Structure Diagram* and a *Context Diagram.* The Structure Diagram is an informal block diagram in which the application is placed relative to lower, higher, and peer-level domains. The utility of the Structure Diagram is to relate the current application, a generic reactive scheduling system, to other applications in the domain. Figure 4 shows the system components in relation to the other pieces of software used to implement the system. Notice that this is the structure diagram required by the domain analysis methodology. The levels identified in this diagram *are not* the reusable layers identified in figure 2. It is possible, and in fact desirable, to map these levels into the four reusable layers previously identified. In section 5.2 the mapping from domain structures levels into reusability layers is made explicit.

Figure 5 presents the Context Diagram for the scheduling problem. The closed boxes in the diagram represent external agents or external source of data; the open boxes represent internal data depositories;

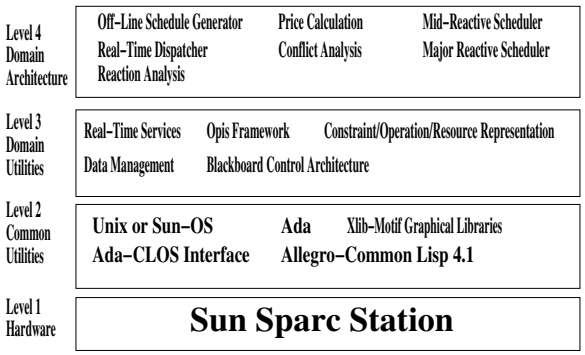

Figure 4: Context Model: Structure Diagram

and the arrows represent data flow.

5 Domain Modeling

The next phase of the domain analysis is the *Domain Modeling.* The is the most important phase for the identification of components at the different levels of the reuse library. The purpose here is to identify the differences and commonalities that characterize the applications in a domain. The three products of the domain modeling are: the *Information Model,* the *Feature Model,* and the *Functional and Behavioral Models.* These products for the domain under

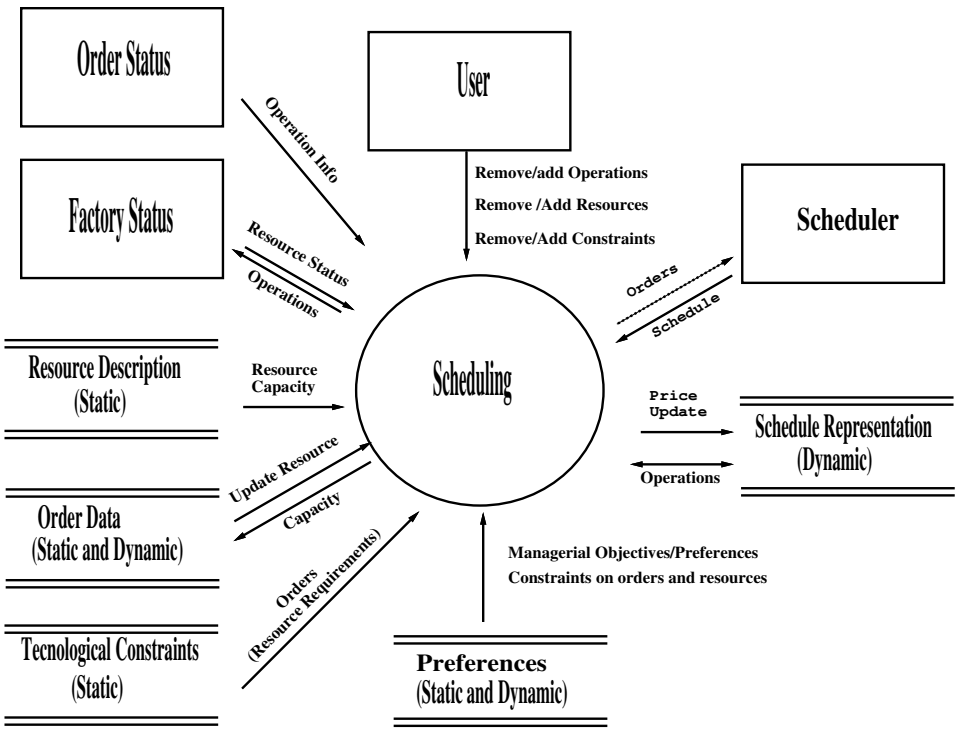

Figure 5: Context Model: Context Diagram

study are too large to be included in this paper[1] but a summary of them is provided.

5.1 Information Model

The *Information Model* captures the domain knowledge and data requirements essential for the development of applications in a domain. This model can be represented as an entity-relationship diagram that specifies the objects to be manipulated by the application and how they relate to each other. The entities provide the needed information for the design of components in library layer 2 – *Domain Information Components.* The relationships between entities specify the requirements for components in layer 1 – *Algorithms and Data Structure.*

The internal representation of objects used by the system is based on the OPIS[17] representation. OPIS design is object-oriented and uses a class hierarchy to represent resources and operations. As described in [13] and [19], its class library can be divided into three main groups:

Base Classes: are usually not instantiable.[2]

Specialized Classes: are instantiable.

Mixin Classes: provide common functionality to different objects.

Figure 6 shows the relation between the three basic class types. From an entity-relationship perspective, the specialized classes define the entities of the system and the base classes and mixin classes define, in a sense, the relationship among them.

In a scheduling system, six base classes can be identified. These base classes correspond to the domain entities and provide the components for library layer 2. The base classes are:

Demand: represents the input or order introduced into the system.

Product: specifies the kind of services provided.

Operation: is the focus of the scheduling system: operations are the entities processed to satisfy the demand.

Resource: represents the entity to be reserved over time to process the operations.

Preference: represent the static knowledge about the system operation. Preferences are used to guide scheduling decisions.

[1] A technical report containing the entire products is available.

[2] These classes are preferably implemented as abstract classes. CLOS, however, does not provide a mechanism to avoid direct instantiation of any class.

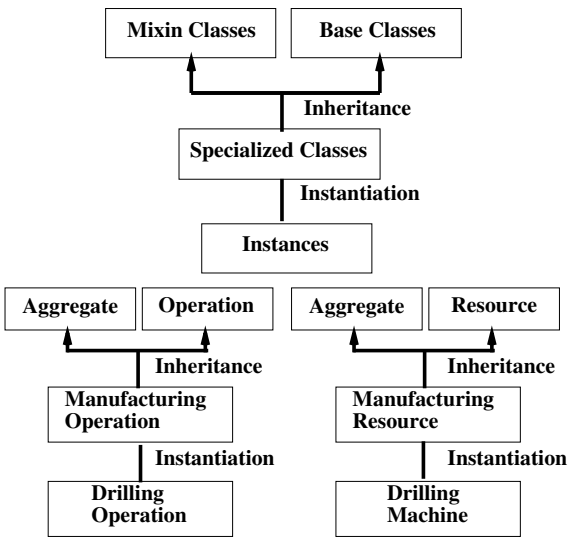

Figure 6: Class Hierarchy

Scheduler: represents the abstraction of the set of functional elements that manipulate demands, resources, operations, and preferences to produce the final schedule.

From the model described above it is possible to define a semantic network specifying the data structures and functionality needed in library layer 2. However, additional elements must be provided to support the functions specified. For example, since operations have start and end times and resources are allocated over time, some kind of mechanism for time services and for manipulating allocation intervals must be provided: time services, representation for time intervals, doubly-linked lists, binary trees, search and sort algorithms, and graphical libraries for interface construction are some of the components needed. As these components do not depend on any kind of domain specific information, they define the components in library layer 1, the domain independent layer.

5.2 Feature Model

The *Feature Analysis* identifies the services provided by the system and how these services differ across applications within the domain. The final product of this phase, the *Feature Model*, is a hierarchical graphical representation of the features. The Feature Model provides the elements to design the components at layer 3.

According to the Feature Oriented methodology, three distinct groups of features are considered:

Display Features: These features specify how the system operation is seen by the user. For a scheduling system, these features include graphs, reports, gantt charts, and tables.

Operational Features: These features specify the general functionality the system should have. The top-level operational features include:

Input: ability of the system to receive information from the external world.

Schedule: ability of the system to assign operations to resources (or vice-versa.) over time.

Dispatch: capability of the system of actually executing the schedule generated.

React: ability of the system to correct the schedule decisions while the schedule is executed.

Context Features: This set of features identifies the dimensions along which the operational and display features show room for adaptability. The top-level context features are:

Application Domain: the kind of operations and resources used in the application domain are an important factor to determine the kind of behavior of the system.

Objective-Function: the managerial objectives defines the scheduling and reaction strategies as well as the price calculation methods.

Level of Abstraction: resources and operations can be aggregated at different levels of abstraction like machines that can be aggregated in work areas, or ships that can be aggregated in fleets.

Schedule Type: the schedule can be *price-based* or *interval-based.*

Strategy: different types of strategy can be established for scheduling (generative or reactive, resource or operation based, etc), reaction, dispatch, analysis, etc.

Conflict: conflicts can differ according to type, size, and importance.

The context features specify the interface between components in layers 2 and 3. To illustrate this notion, consider the base classes `operation` and `resource` defined as components in library layer 2. As it was mentioned before, base classes cannot be instantiated. In level 3, the specialized classes `manufacturing-operation`, `manufacturing-resource`, `transport-operation`, and `transport-resource` are created as sub-classes of the corresponding base classes. The context feature responsible for the differentiation of the components in this example is the application domain. In the appendix a more detailed example illustrating this situation is presented.

5.3 Functional Model

The *Functional Analysis* identifies the control structure and data flow necessary to implement the services described in the feature analysis. The product of the Functional Analysis, the *Functional and Behavioral Model,* captures functional commonalities and parameterizes variability. This model specifies how

the components at layer 3 can be parameterized to provide the specific functionality described in the Feature Model.

Once specialized classes and methods have been defined at layer 3 of the component library, instances of the specialized classes can be created at layer 4 and methods can be combined to provide the required functionality. The components of the application, layer 4, are specified based on the operational features, and parametrized based on the context features. For example, a scheduling method defined at layer 4 can be a combination of several different search strategies defined at layer 3. The context feature *Application Domain* establishes the type of operations and resources that will be used by the scheduling methods; the context feature *Objective Function* establishes the search strategy that should be selected to generate the schedule; the context feature *schedule type* determines if time bound intervals should be propagated each time a decision is made or if a price computation method should be triggered.

Notice that the context features identify the specialization from base classes to specialized classes and also guide the class selection for instantiation. This ambiguity in the role of the context features is one of the problems we found in using the Feature Oriented methodology: although it helps the identification of the desired functionality (operational features) and its variability (context features), it does not provide a means of directly relating them. The connection between the different phases is also left unspecified: the methodology does not provide any information on how the operational features should manipulate the objects defined in the information model or how to use the feature model to generate the functional model.

To implement the general system behavior described before, five semi-finished components can be identified. These five components and respective functionality are:

Top-Level Manager: problem-solving agents coordination.

Real-Time Dispatcher: scheduled operations execution.

Conflict Analyzer: conflict and reaction analysis.

Reaction Agent: schedule repair. The size of the reaction is a function of the conflict characteristics.

Price Calculation: priority list computation. The priorities define the operation sequence for execution.

The general behavior of the system can be seen in the Functional Diagram presented in figure 7. The Functional Diagram represents the activity flow of the system. The larger boxes represent the architectural components in layer 4 of the library. Each of these components is implemented by the composition of instances of semi-finished components from layer 3. The small boxes (except the "idle" boxes) represent instances of layer 3 components. For example, when orders are introduced, the *Read-Orders* element should be able to read the input and generate the internal representation of these orders. The Read-Orders functionality is obtained by the instantiation of two semi-finished components: one that will read the orders introduced in the system and other that will translate the input read into the internal representation. The semi-finished component that generates the internal representation for the input is the *Order Instantiator.*[3] This element corresponds to an instance of the class *Instantiator* described in library layer 3 as a specialization of the corresponding layer 2 base class. Notice that the final instantiated component hides the inner lower level relations.

The orders and respective operations are then sent to the element in charge of schedule generation and price computation. The layer 4 component *Generate Schedule* is obtained by the instantiation of three semi-finished components: the schedule generation element corresponds to an instance or a combination of instances of semi-finished components identified as *problem-solvers;* the price computation is performed by a price calculation instance that is parameterized by the objective function that the scheduler is trying to optimize; the dispatch list is then generated by an instance of a dispatch list generator.

The output of the *Generate-Schedule* element is a priority list that is sent to the real-time dispatcher. The real-time dispatcher is a component in layer 4 that will use one or more of the dispatch algorithms defined in layer 3 and, like the *Schedule Instantiator,* will modify instances of the specialized classes for resources and operations. Different dispatch algorithms are used depending on the set of context features selected for this particular application. The dispatcher operates in a cycle, and as conflicts are detected, they are sent to the element in charge of mid-size corrections. If the mid-size reaction cannot solve the problem, a global repair is performed.

6 Domain Library Implementation

Architectural Modeling is the last phase of the Feature Oriented methodology. The product of this phase is a software solution in the form of a high-level design of the application. Instead of presenting architectural modeling as specified by the methodology, this subsection presents how the components defined in library layers 1, 2, and 3 can be used to generate an application at layer 4.

As it was said before, it is possible to map the products of the different phases of the methodology to the design levels of the components. Figure 8 shows how the Structure Diagram of figure 4 maps into the four design layers of the components. The Common Utility

[3] Do not confuse the *Order Instantiator* with the instantiation of semi-finished components. From the scheduling system point of view, the *Order Instantiator* is a component responsible for creating instances of operations and demands. From the library perspective, it is an instance of a semi-finished component.

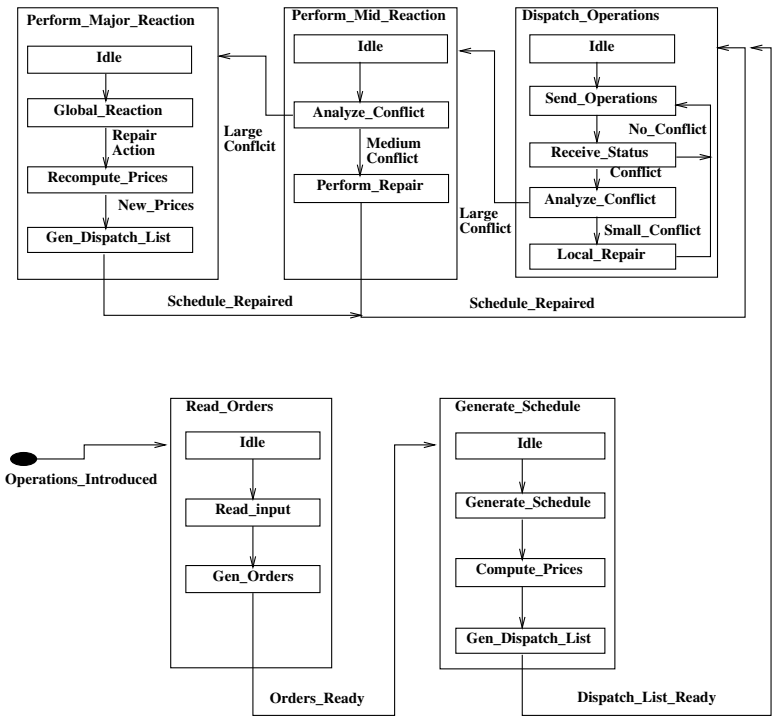

Figure 7: Functional Model: Functional Diagram

level of the Structure Diagram corresponds to library layer 1 – Domain Independent Components. The components at this layer provide the most basic services like time manipulation routines, balanced trees, doubly linked lists, and graphical libraries. These data structure definitions and respective manipulation routines can be found in public software repositories or in several textbooks.

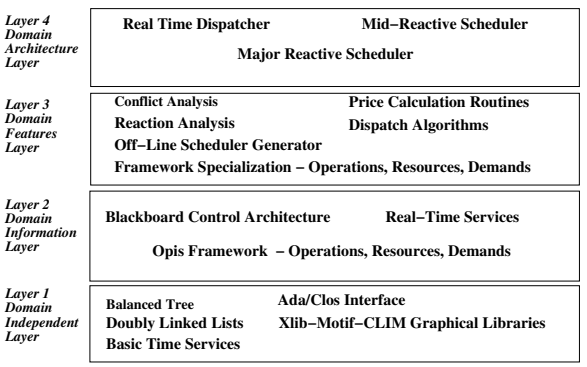

Figure 8: Design Levels for Reusable Components

The Domain Utility level in the Structure Diagram corresponds to library layer 2 – Domain Information Layer. At this layer there are routines to implement the control cycle, the instantiation of external demands into internal orders, the time bound propagation mechanism, and the basic representation for demands, products, operations, and resources. The components at this level can be implemented from scratch based on the *Information Model* or can be retrieved from existing systems by using re-engineering approaches[1, 6].

The Domain Architecture level of the Structure Diagram maps into layers 3 and 4. At layer 3 is the specialization of the framework for the specific application domain. Operations and resources are specialized so that they fulfill the needs of specific requirements. In layer 3 there are also the domain dependent algorithms for conflict and reaction analysis as well as dispatch heuristics.

Library layer 4 corresponds to the application that uses the components defined in layer 3. All the elements at the topmost layer are implemented by selecting, instantiating, and coordinating the functions provided by the features defined in layer 3.

7 Conclusions

This paper deals with several different issues related to software reusability. The definition of these issues clarifies the objectives for the designer. Instead of aiming for a universal remedy, we suggest that reuse entails a fundamental conflict between the intentions of the original component designer and the reusing de-

veloper. This conflict can be resolved only by applying a step-wise reduction to the gap separating design-for-reuse from design-with-reuse. Design techniques play an important role in shaping the static structure of the resulting component library.

The approach presented here is based on the proposition that the formalization of the software library static structure serves to bridge the gap between the component library and the software architecture.

The first contribution of the present work is the testing and validation of a domain analysis methodology as a useful tool for the identification of commonalities and differences among related applications for the same kind of domain. The second contribution is the construction of the mapping from the output products of the Feature Oriented Methodology to a design technique for reusable components that establishes several levels of reusability. The third contribution is the validation of the design technique as an efficient mechanism for the generation of domain dependent reusable software components.

Validating the concept by applying it in a professional environment and collecting the opinions of the users is one avenue for further work.

Parallel to these contributions to the software engineering field, there are the contributions related to scheduling and knowledge-based applications. The by-products of using these two methodologies are a better understanding of the scheduling domain and a library of reusable components for the implementation of scheduling systems to be used in different applications.

The Feature Oriented Methodology has been a useful tool for the identification of the desired system functionalities and the dimensions along which they can vary. The main difficulty we found was how to establish the link between the different phases. The methodology does not specify how to generate a functional model or the architectural model from the feature and information model. To deal with these issues, the solution was to adapt the products of the Feature Oriented methodology so they could be used as an input to the design phase of another methodology.

We have been also looking at specification languages that allow domain description in higher level structured languages. The time when a programming language possess features for enforcing guidelines is not near, and it might never be. Management and engineering practices must compensate for this lack of language support. Enforcing the use of components from the next lower layer in the model is one rule that management has to supervise. Configuration management tools can provide desirable features like making different bodies available for a given specification. Standard naming schemes and powerful library management tools are other approaches for reducing reuse difficulties.

As it was mentioned before, the library of components resulting from the analysis described has been partially implemented by re-engineering parts of existing scheduling systems. Considering the architecture described in section 5.3, we are currently working on the implementation of some components at layer 3 and 4, namely the analysis and price computation agents. The current system implementation is in Common Lisp Object System[20] but we are considering the re-implementation of the entire system in Ada9X since CLOS does not allowed the layering of components. The results of the domain analysis is also being currently used to re-engineer another scheduling system for vehicle movement control.

Acknowledgments

Many thanks to Shalom Cohen and James Withey of the SEI for their expert advice and *coaching* on the use of the Feature Oriented Domain Analysis methodology and also to Ora Lassila from CIMDS for his help and suggestions.

References

[1] V.R. Basili, "Reusing existing software," *Technical Report UMIACS-TR-88-72,* Dept. of Computer Science, University of Maryland, College Park, MD, Oct 1988.

[2] M.A. Becker, "Revision of reactive scheduling architectures," *Working Paper,* Robotics Institute, Carnegie Mellon University, Pittsburgh 1993.

[3] M.A. Becker, "A price-based reactive scheduling architecture," *Working Paper,* Robotics Institute, Carnegie Mellon University, Pittsburgh 1994.

[4] G. Booch, *"Software Components with Ada,"* Benjaming/Cummings, Menlo Park, CA. 1987.

[5] E.V. Berard, "GRACE Software Components," *EVB Software Engineering,* Frederick, MD, 1986.

[6] G. Caldiera and V.R. Basili, "Reengineering existing software for reusability," *Technical Report UMIACS-TR-90-30,* Dept. of Computer Science, University of Maryland, College Park, MD, Feb 1990.

[7] S.G. Cohen, J.L. Stanley, A.S. Peterson, and R.W. Krut, "Application of feature-oriented domain analysis to the army movement control domain," *SEI-91-TR-28,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1992.

[8] J.L. Díaz-Herrera, M. Schumacher, and M.A. Becker, "Institutionalizing Software Reuse: bridging the gap between Design-with-reuse and Design-for-reuse," *Submitted for publication,* 1994.

[9] J.A. Hess, W.E. Novak, P.C. Carrol, S.G. Cohen, R.R. Holibaugh, K.C. Kang, and A.S. Peterson, "A domain analysis bibliography," *SEI-90-SR-3,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 90.

[10] R.E. Johnson and B.Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming,* June/July, pp 22-35 (1988).

[11] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, "Feature-oriented domain analysis: Feasibility study," *SEI-90-TR-21,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[12] R.W. Krut Jr, "Integrating 001 tool support into the Feature Oriented Domain Analysis Methodology," *SEI-93-TR-11,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[13] O. Lassila, "Object Oriented Description of OPIS," *Internal Report,* Robotics Institute, Carnegie Mellon University, July 93.

[14] Levy & Ripken, *Proceedings of the 1987 Ada-Europe conference,* pp 100-112

[15] E. Mettala, and M.H. Graham, "The Domain-Specific Software Architecture Program," *CMU/SEI-92-SR-9, ADA257225,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1992.

[16] T. Morton, *Heuristic Scheduling Systems,* GSIA, Carnegie Mellon Univ., Pittsburgh,PA, July 1992.

[17] S.F. Smith, " The OPIS framework for modeling manufacturing systems," *Technical Report CMU-RI-TR-89-30,* The Robotics Institute, Carnegie Mellon University, December 1989.

[18] S.F. Smith, "Knowledge-based production management: approaches, results and prospects," *Production Planning and Control,* vol. 3, no. 4, pp. 350-80, 1992.

[19] S.F. Smith and O. Lassila, "Configurable systems for reactive production management," *Working Paper.* CIMDS, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Aug. 93.

[20] G.L. Steele Jr.,*Common Lisp* 2nd Edition, Digital Press, 1990.

Appendix: Components in CLOS

In a scheduling system, two basic entities are *operations* and *resources.* In CLOS (Common Lisp Object System), these two entities can be defined as:

```
(defclass operation ()
 ((duration)
  (start-time)
  (end-time)
  (resource  ))

(defclass resource ()
  ((resource-capacity)
   (scheduled-operations)
   (unscheduled-operations))
```

From the context features presented in section 5.2 it is possible to see that the application domain is a dimension along which the components should differ. In the manufacturing domain, to compute the duration of an operation, it is important to consider the duration of the machine setup. The representation of resources should also keep a record of incomplete operations due to machine or operator failures. In the transportation domain, it would be useful to represent the origin and destination of each operation, and the transportation resource should also specify the velocity and location of the resource. In level 3, the classes `operation` and `resource` are specialized for the manufacturing and transportation domain as:

```
(defclass manufacturing-operation
  (operation)
  ((setup-duration)))

(defclass transport-operation
  (operation)
  ((origin)
   (destination)))

(defclass manufacturing-resource
  (resource)
  ((incomplete-operations)))

(defclass transport-resource
  (resource)
  ((velocity)
   (location)))
```

The same can be done with methods or functions. Using still the same example, consider the duration of the operations:

For the base class `operation` the method `operation-duration` reads the value of the duration directly from the slot duration. For manufacturing-operations, however, it is necessary to add the setup duration to the duration of the operation itself. A method to do this could be written as:

```
(defmethod operation-duration
 ((op manufacturing-operation))
 (+ (setup-duration op)(duration op)))
```

Where `duration` is a method that will compute the individual duration of the operation represented by `op`.

In the transportation domain, the duration of the trip is a function of the distance between origin and destination and the velocity of the resource:

```
(defmethod operation-duration
 ((op transport-operation))
 (let ((resource (operation-resource op)))
     (/ (distance-between
        (origin op)(destination op))
        (velocity resource))))
```

Where `distance-between` is a function which, given two locations, returns the distance between them and `operation-resource` is the method that reads the value of the slot `resource` of the operation.