Technical Report

CMU/SEI-90-TR-21
ESD-90-TR-222

Feature-Oriented Domain Analysis (FODA) Feasibility Study

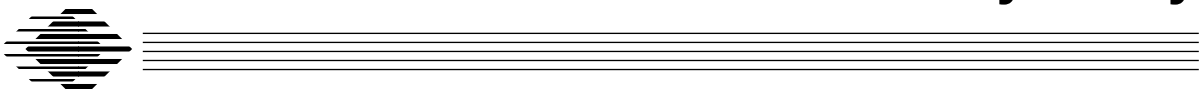
Kyo C. Kang
Sholom G. Cohen
James A. Hess
William E. Novak
A. Spencer Peterson

November 1990

Technical Report

CMU/SEI-90-TR-21
ESD-90-TR-222
November 1990

Feature-Oriented Domain Analysis (FODA) Feasibility Study



Kyo C. Kang
Sholom G. Cohen
James A. Hess
William E. Novak
A. Spencer Peterson

Domain Analysis Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Acknowledgements

We would like to acknowledge the support of Capt. Patrick Carroll, USAF, for his contribution to the project, which helped make this report possible.

We would also like to acknowledge the support and expert counsel of Bill Wood, Mario Barbacci, Len Bass, Robert Firth, and Marc Kellner of the SEI for helping to shape and clarify the ideas contained within. Thanks go also to Brad Myer of Carnegie-Mellon University for sharing his domain expertise in window management systems.

Finally, we would like to thank the careful reviewers of a large document, specifically Len Bass, Robert Firth, Larry Howard, Tom Lane, Nancy Mead, Don O'Neill, Dan Roy, and Kurt Wallnau.

Executive Summary

This report describes a method for discovering and representing commonalities among related software systems. By capturing the knowledge of experts, this *domain analysis* method attempts to codify the thought processes used to develop software systems in a related class or *domain*. While domain analysis supports software reuse by capturing domain expertise, domain analysis can also support communication, training, tool development, and software specification and design.

The primary focus of the method is the identification of prominent or distinctive *features* of software systems in a domain. These features are user-visible aspects or characteristics of the domain. They lead to the creation of a set of products that define the domain and also give the method its name: *Feature-Oriented Domain Analysis* (FODA). The features define both common aspects of the domain as well as differences between related systems in the domain. Features are also used to define the domain in terms of the mandatory, optional, or alternative characteristics of these related systems. This report provides a description of the products of the domain analysis, as well as the process for generating them. The report also contrasts the FODA method with other related work.

An important component of this report is a comprehensive example of the application of the method. The example demonstrates the utility of the FODA method in providing an understanding of a domain, both in terms of the scope of the domain and in terms of the features and common requirements. The report also describes several technical issues raised during the development of the method. These issues will be further explored in subsequent domain analyses.

This report does not cover the non-technical issues related to domain analysis, such as legal, economic, or managerial issues. The emphasis on defining process and products stems from the belief that the non-technical issues can be fully explained only in light of specific approaches to domain analysis and to reuse in general. This report establishes one such approach.

Domain analysis remains a relatively new practice. Although first proposed ten years ago, domain analysis is still a topic primarily of research groups. The report *Feature-Oriented Domain Analysis* can advance the state of the practice of domain analysis by providing meaningful examples and issues for further exploration.

Unix is a trademark of AT&T Bell Laboratories.

Open Look is a trademark of AT&T.

OSF/Motif is a trademark of the Open Software Foundation.

The X Window System is a trademark of the Massachusetts Institute of Technology.

SunView and NeWS are trademarks of Sun Microsystems, Inc.

VMS Windows is a trademark of Digital Equipment Corporation.

Macintosh is a trademark of Apple Computer, Inc.

Symbolics is a trademark of Symbolics, Inc.

Statemate is a trademark of i-Logix, Inc.

Feature-Oriented Domain Analysis

Abstract: Successful software reuse requires the systematic discovery and exploitation of commonality across related software systems. By examining related software systems and the underlying theory of the class of systems they represent, domain analysis can provide a generic description of the requirements of that class of systems and a set of approaches for their implementation. This report will establish methods for performing a domain analysis and describe the products of the domain analysis process. To illustrate the application of domain analysis to a representative class of software systems, this report will provide a domain analysis of window management system software.

1. Introduction

1.1. Scope

The systematic discovery and exploitation of commonality across related software systems is a fundamental technical requirement for achieving successful software reuse [Prieto-Diaz 90]. Domain analysis is one technique that can be applied to meet this requirement. By examining a class of related software systems and the common underlying theory of those systems, domain analysis can provide a reference model for describing the class. It can provide a basis for understanding and communication about the problem space addressed by software in the domain. Domain analysis can also propose a set of architectural approaches for the implementation of new systems.

The primary intent of this report is to establish the *Feature-Oriented Domain Analysis* (FODA) method for performing a domain analysis. The *feature-oriented* concept is based on the emphasis placed by the method on identifying those features a user commonly expects in applications in a domain. This method, which is based on a study of other domain analysis approaches, defines both the products and the process of domain analysis. The report also provides a comprehensive example to illustrate the application of the FODA method to a representative class of software systems.

This report is directed toward three groups:

1. Individuals providing information about a domain under analysis (domain experts).
2. Individuals performing the domain analysis (domain analysts).
3. Consumers of domain analysis products (systems analysts and developers).

The roles of each group with respect to domain analysis will be defined later in this chapter.

This report provides an introduction to domain analysis, a description of a method for per-

forming the analysis, and a sample application of that method. It does not address such related areas as legal and economic issues pertaining to domain analysis, nor does it explore other non-technical areas.

The report is organized in the following way:

- The rest of this chapter will introduce the concepts of domain analysis
- The second chapter provides an historical overview of the field, describing several methods proposed for performing domain analysis; the chapter also establishes a set of basic criteria for evaluating domain analysis methodologies.
- Resulting from this evaluation of existing methodologies, Chapter 3 presents an overview of the Feature-Oriented Domain Analysis (FODA) method by describing the three basic activities of the method: *context analysis*, *domain modelling* and *architecture modelling*. Chapters 4-6 then contain detailed discussions of each of these activities and the products produced.
- Chapter 7 presents a domain analysis of window manager software, illustrating the application of the FODA method.
- The appendices provide the detailed domain terminology dictionary and other supporting information to document the window manager software domain analysis. Other material supporting the method is also presented in the appendices.

1.2. Domain Analysis Concepts

The development of large and complex software systems requires a clear understanding of desired system features and of the capabilities of the software required to implement those features. Software reuse, which has long promised improvements in the development process, will become feasible only when the features and capabilities common to systems within a domain can be properly defined in advance of software development. Domain analysis, the systematic exploration of software systems to define and exploit commonality, defines the features and capabilities of a class of related software systems. Thus, the availability of domain analysis technology is a factor that can improve the software development process and promote software reuse by providing a means of communication and a common understanding of the domain.

The list below offers definitions of several terms which are basic to domain analysis, and which are essential to the following discussion of a domain analysis method.

Application: A system which provides a set of general services for solving some type of user problem.

Context: The circumstances, situation, or environment in which a particular system exists.

Domain: (also called *application domain*) A set of current and future applications which share a set of common capabilities and data.

Domain analysis: The process of identifying, collecting, organizing, and

	representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain.
Domain engineering:	An encompassing process which includes domain analysis and the subsequent construction of components, methods, and tools that address the problems of system/subsystem development through the application of the domain analysis products.
Domain model:	A definition of the functions, objects, data, and relationships in a domain.
Feature:	A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [American 85].
Software architecture:	The high-level packaging structure of functions and data, their interfaces and control, to support the implementation of applications in a domain.
Software reuse:	The process of implementing new software systems using existing software information.
Reusable component:	A software component (including requirements, designs, code, test data, etc.) designed and implemented for the specific purpose of being reused.
User:	Either a person or an application that operates a system in order to perform a task.[1]

Because it is central to several of the definitions and to the concept of domain analysis itself, the term *domain* requires some clarification through examples. A domain does not necessarily have to occur at a specific level of software granularity, such as that of a system, Computer Software Component (CSC), or Computer Software Configuration Item (CSCI). Rather, a domain is a more general concept which may be stretched to apply to most any potential class of systems. This class is referred to as the *target domain,* which may have both higher-level domains to which it belongs and *sub-domains* within it. As an example, different instances of the same type of system (such as window management systems or relational database management systems) can be grouped together to define a domain. In a similar way a domain of data structures could be identified which would be at a much lower level than that of entire systems, but could still constitute a target domain in its own right. Grady Booch could be said to have performed an analysis of this domain which resulted in the creation of the abstract data type components described in [Booch 87]. In another context this target domain of abstract data types could be viewed as a sub-domain of the larger data management domain. Put another way, the X window system is correctly thought of as a system on which user application programs are written and executed. From a different perspective, however, a collection of similar window management systems such

[1]The *user* of a system is not necessarily the same as the *customer* for a system. These are two separate concepts, although they may be combined in many cases.

as X windows, NeWS, and others, constitutes a domain, and each window management system is an application in that domain. Both views are correct, in that X windows is both a complete system and an instance of a larger domain. One view may be more useful than the other if it helps to solve the problem at hand.

1.2.1. Domain Analysis Process

Domain analysis gathers and represents information on software systems that share a common set of capabilities and data. The methods proposed in [GILR89A, MCNI86A, PRIE86A] suggest that there are three basic phases in the domain analysis process:

1. *Context analysis:* defining the extent (or bounds) of a domain for analysis.
2. *Domain modelling:* describing the problems within the domain that are addressed by software.
3. *Architecture modelling:* creating the software architecture(s) that implements a solution to the problems in the domain.

Figure 1-1 depicts the three groups of participants in the domain analysis process. During each phase these players take on slightly different roles.

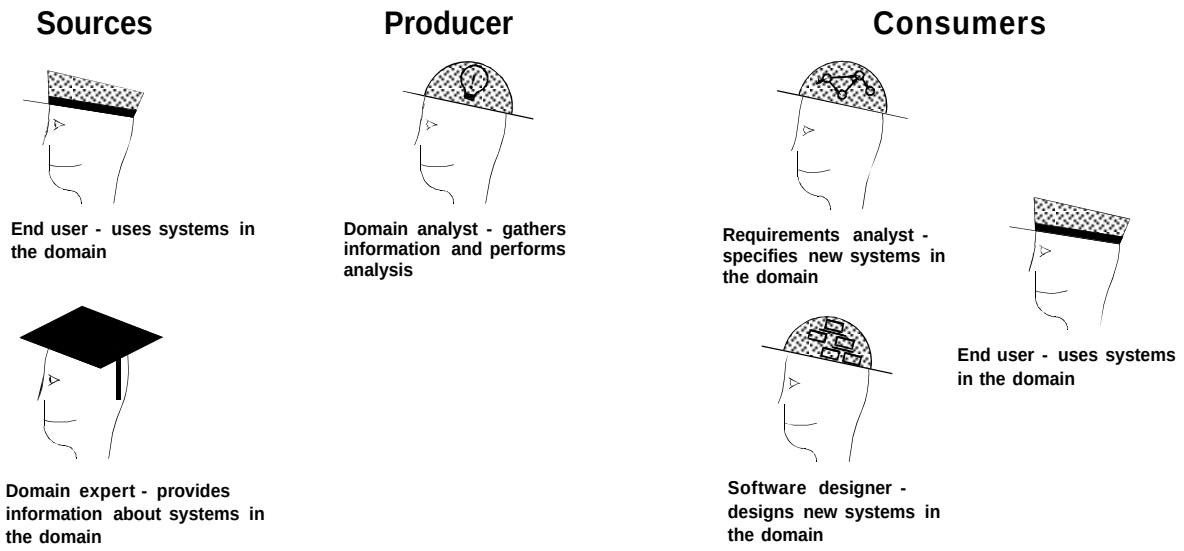


Figure 1-1: Participants in the Domain Analysis Process

- Context analysis: The domain analyst interacts with users and domain experts to establish the bounds of the domain and establish a proper scope for the analysis. The analyst also gathers sources of information for performing the analysis.
- Domain modelling: The domain analyst uses information sources and the other

products of the context analysis to support the creation of a domain model. This model is reviewed by the system user, the domain expert, and the requirements analyst.

- Architecture modelling: Using the domain model, the domain analyst produces the architecture model. This model should be reviewed by the domain expert, the requirements analyst, and the software engineer. The user need not participate in this review.

The requirements analyst and software designer will use the products of a domain analysis when implementing a new system. During implementation, these products are tailored to produce the specification and design of that system. This tailoring process will provide feedback of information to the domain analyst and expert to modify or extend the domain analysis for future developments. (See Figure 1-2.) This feedback may also serve to improve the domain analysis process, by discovering possible weaknesses in the original methods. The specific roles of the participants in the tailoring process will be defined as part of the FODA method in Chapter 3.

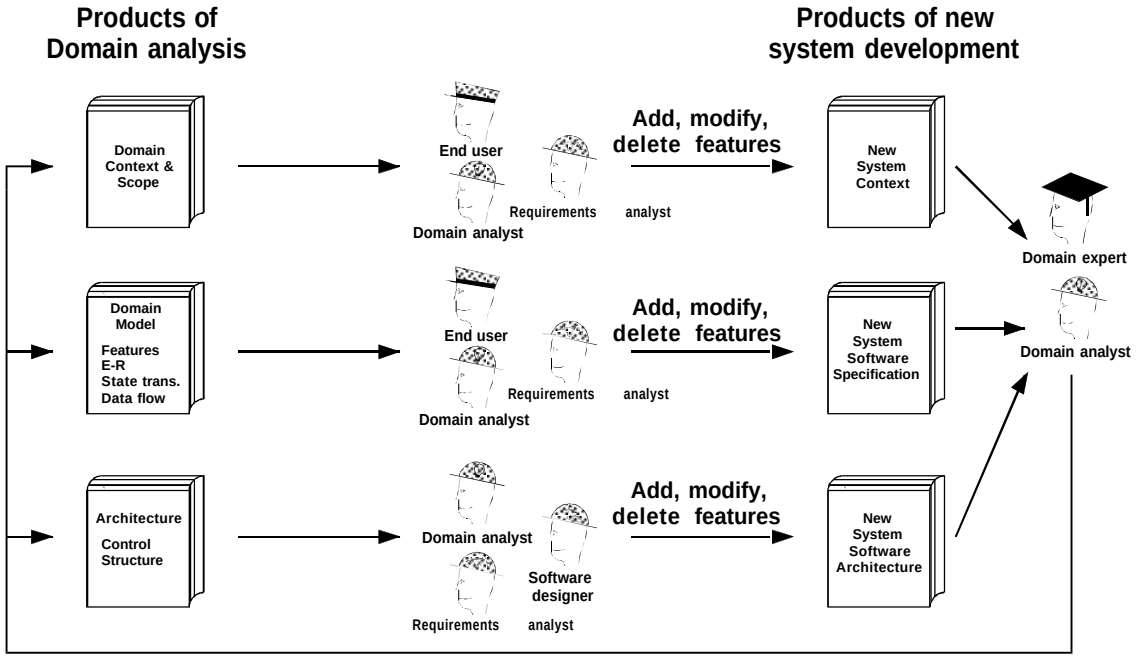


Figure 1-2: Tailoring the Products to Enhance the Domain Analysis

1.2.2. Domain Analysis Products

The domain analysis method must provide specific representations to document the results of each of the domain analysis activities. These representations form a reference model for systems in the domain. The representations define the scope of the domain, describe the problems solved by software in the domain, and provide architectures that can implement solutions.

For each of the three phases of the domain analysis process there will be a separate set of representations of the domain.

Context analysis: The results of this phase provide the context of the domain. This requires representing the primary inputs and outputs of software in the domain as well as identifying other software interfaces.

Domain modelling: The products of this phase describe the problems addressed by software in the domain. They provide:

- features of software in the domain
- standard vocabulary of domain experts
- documentation of the entities embodied in software
- generic software requirements via control flow, data flow, and other specification techniques

Architecture modelling: This phase establishes the structure of implementations of software in the domain. The representations generated provide developers with a set of architectural models for constructing applications and mappings from the domain model to the architectures. These architectures can also guide the development of libraries of reusable components.

Figure 1-3 depicts the three phases of the method and lists the products of each.

The FODA method provides a means of applying these products of domain analysis to support software development. Figure 1-4 depicts the use of domain analysis products and their ability to provide a means to support communication between users and developers. This view of domain analysis shows that it can be integrated into a more general process for software development to support:

- understanding the domain
- implementing applications in the domain
- creating reusable resources (designs, components, etc.)
- supporting creation of domain analysis and other reuse tools

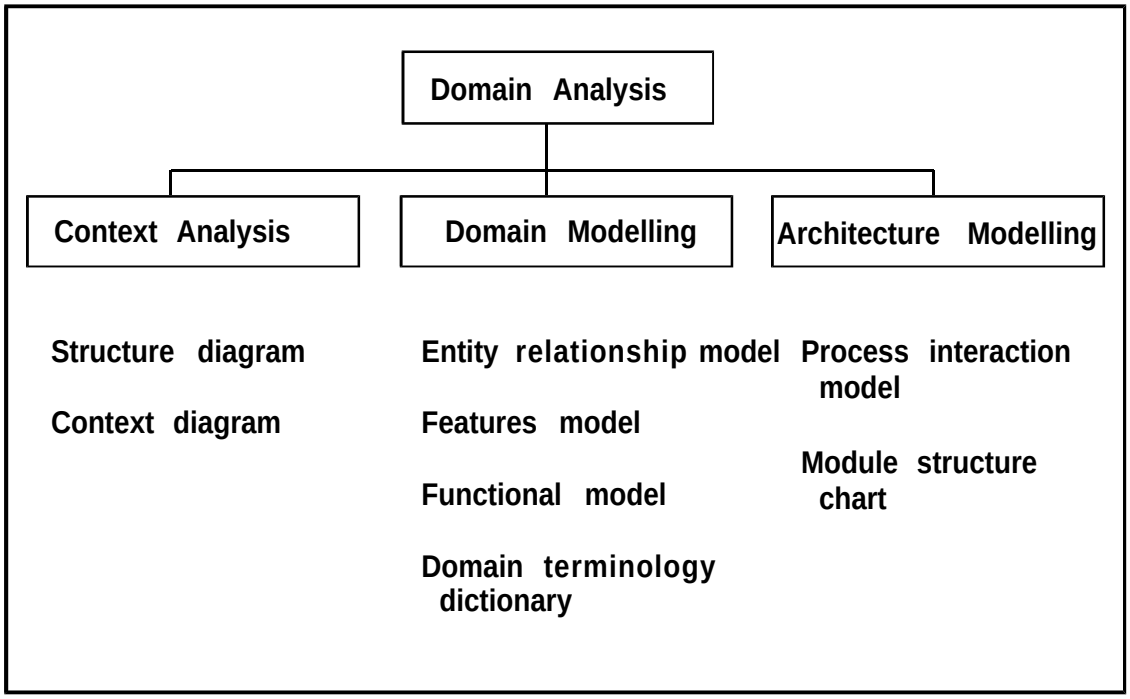


Figure 1-3: Phases and Products of Domain Analysis

1.3. Feasibility Study Overview

Before a discussion of the details of the FODA domain analysis method itself is appropriate, it is necessary to discuss the context in which this feasibility study was performed. Certain constraints applied to this initial study influenced the work, and are re-examined in Chapter 8 in light of the study's results.

First, domain analysis is still a research topic. Despite the different efforts outlined in Chapter 2 there is no uniform agreement on method, representation, or products. This report presents a proposed approach and some experience in applying it, but does not attempt to imply that all of the central issues surrounding domain analysis have been resolved.

Second, the application of the FODA method to the window manager domain was done as a feasibility study to see if it would be possible and useful to analyze application domains with this method. While most aspects of the method were applied to the sample window manager domain, it was not deemed necessary to exhaustively apply the method beyond the point where basic feasibility had been determined and significant lessons learned.

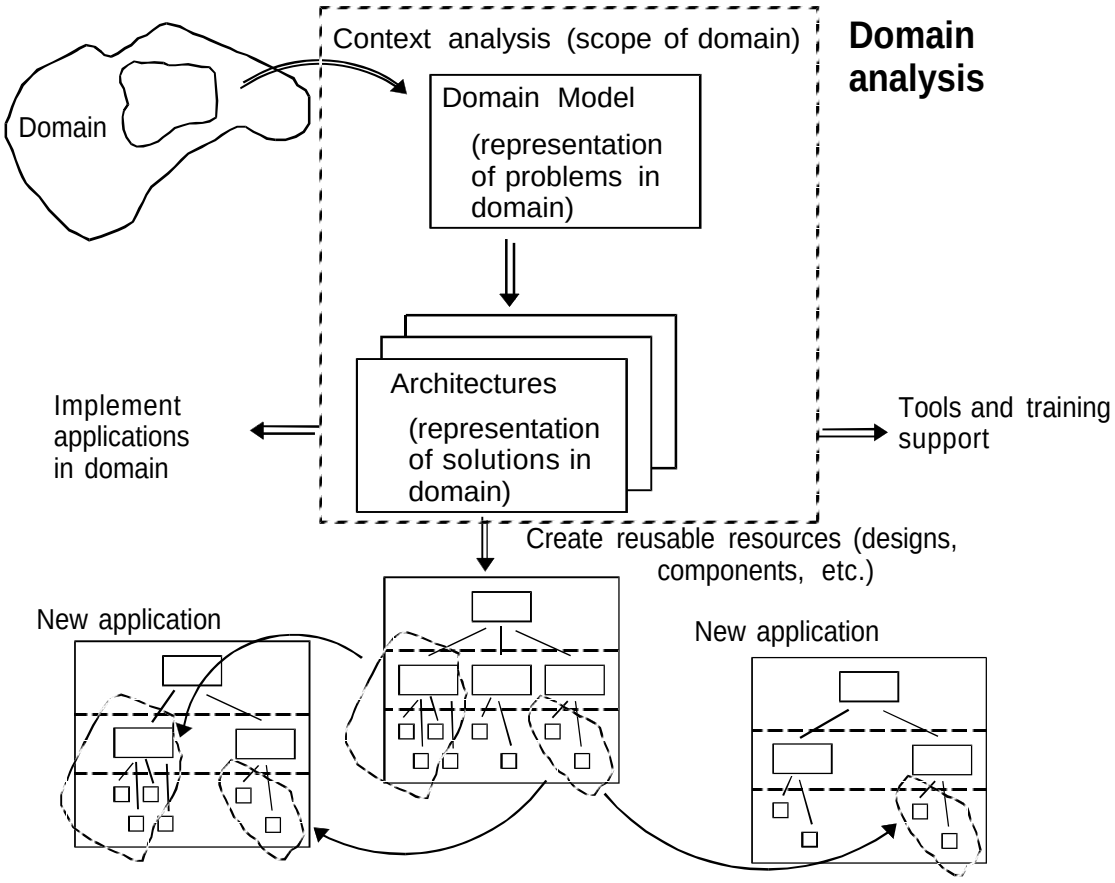


Figure 1-4: Domain Analysis Supports Software Development

Third, in performing the sample domain analysis no sufficiently mature automated tool support for domain analysis was available. While general purpose tools are available which can support some domain analysis functions, and prototype tools have been built specifically to support domain analysis activities, no tool support was available which was both robust and specific to domain analysis. In addition, the purpose of the study was to demonstrate the feasibility of a general domain analysis method, rather than the effectiveness of any particular support environment. As a result, primarily manual methods were used, with some specific automated support such as Statemate for some of the model types. As is discussed in Section 8.1.3, the issue of effective knowledge representation will be a focus of future work.

Fourth, at the time the feasibility study began, the definition of the third and final phase of the FODA method, architecture modelling, had not been completed. Therefore, while the general approach to this phase is defined in this document, it was not applied to the sample domain analysis. One effect of this is that the architecture modelling phase of the method is not as specific in direction as the others because there has been no feedback to it from actual use.

Fifth, the products of the sample window manager domain analysis were not formally validated against another application in the domain (one which was not used as an input to the domain analysis), as the FODA method recommends. The products were informally validated through review by experts, and showed good agreement across the eight window manager systems which were used as inputs to the analysis.

1.4. Successful Application of Domain Models

The process of domain analysis has been successfully applied in many mature domains. While not formally called domain analysis, companies that establish generic architectures for software systems that they build are creating and using some of the products of a domain analysis. This is precisely the approach taken by the Toshiba Corporation [Matsumoto 84], which has successfully established a software "factory" that can produce many highly similar systems that are customized to the specific needs of each customer. The Thomson-CSF Company [Andribet 90] has also used this approach to develop air traffic control systems. They recognize that they are building similar software systems in an established domain, so the architecture becomes a standard from which new systems are derived. This is a variation of domain analysis.

These successes point towards the need for a domain analysis method to achieve two specific goals:

1. The method leads to the development of domain analysis products that support implementation of new applications. This goal will be met when domain analysis products are used in new implementations.
2. The method can be incorporated into the general software development process. This goal will be met when domain analysis methods become an accepted part of software development.

The successful application of methods similar to those proposed in this report support the contention that the products of a domain analysis can be used efficiently to produce new systems in the domain, and to implement software reuse.

The use of the FODA method in this feasibility study, while successful in explicitly setting forth the capabilities of systems in the domain, is not yet a complete success for the method. The method produces accurate products which describe the domain, but these products have not been used in the implementation of new applications. When this has been done, then the method may be considered a success.

The next chapter will highlight other domain analysis approaches, document their successes, and describe where they differ from the FODA method. These successes show that domain analysis methods can succeed when:

- The domain of applicability is of suitable scope (i.e., the extent or size of the analysis is feasible).
- The analysis attempts to abstract the requirements to the problem level from the application level.

- The products provide documentation of the problem abstraction and guidance in tailoring the abstraction to meet specific requirements.

This report will provide specific methods to meet these three criteria.

2. Review of Domain Analysis Methods

While the reuse community has not established a uniform approach to identifying reusable software resources, there is general agreement that domain analysis offers the ability to identify and support development of these resources. During the past ten years, there have been several major efforts in domain analysis. Reports on these efforts include descriptions of methods, case studies, and tool recommendations.[2]

The following list provides a brief chronology of those domain analysis studies that describe usable products to support software reuse.

- 1979: Raytheon Missile Systems Division [Lanergan 79]
- 1980: Neighbors' dissertation: *Software Construction Using Components* [Neighbors 80]
- 1985: McDonnell Douglas: Common Ada Missile Packages (CAMP) [McNicholl 86, McNicholl 88]
- 1985: Schlumberger: Domain Specific Automatic Programming [Barstow 85]
- 1988: Batory: Domain Analysis of Database Management Systems [Batory 88a, Batory 88b, Batory 88c]
- 1988: CTA studies and tools for NASA [Bailin 88, Moore 89, Bailin 89]
- 1988: SEI: *An OOD Paradigm for Flight Simulators* [Lee 88, D'Ippolito 89]
- 1989: MCC: DESIRE System [Biggerstaff 89a]
- 1989: Thompson-CSF: Air Traffic Control Systems Domain Analysis [Andribet 90]
- 1989: CSC: Domain Analysis for Flight Dynamics Applications

In addition to the product-directed studies, there have been other studies that focused on the process of domain analysis:

- 1987: Prieto-Diaz: "Domain Analysis for Reusability" [Prieto-Diaz 87]
- 1988: Arango: thesis and other domain analysis studies [Arango 88a, Arango 88b, Arango 88c, Arango 89]
- 1988: Bruns and Potts: "Domain Modeling Approaches to Software Development" [Bruns 88]
- 1988: Lubars: "A Domain Modeling Representation" [Lubars 88]
- 1989: SPS: *Impact of Domain Analysis on Reuse Methods* [Gilroy 89]
- 1990: SPC: *A Domain Analysis Process* [Jaworski 90]

The distinction between product and process emphasis in these lists is not purely organizational; most studies to date have concentrated on one of the two areas. Studies such as those by Neighbors and CAMP describe their products in detail, but give little insight into the

[2]A more complete enumeration of related studies and papers is available in a companion report, *A Domain Analysis Bibliography*, CMU/SEI-90-SR-3 [Hess 90].

methods used to generate them. The process studies such as those by Lubars or SPS define methods but do not give significant examples of the application of those methods. While much work has been done on methods for domain analysis there is no single definitive domain analysis method. The next section proposes some criteria for evaluating domain analysis methods and applies those criteria to support a comparison of existing methods to the FODA method.

2.1. Evaluation of Methods

Gathering and representing domain knowledge in the form of a domain model and generic architecture presents problems that have not been addressed by traditional development methods. These traditional methods are not reuse-based, but rather are oriented toward single system development. Domain analysis can be distinguished from this single system approach in that its objective is to represent exploitable commonalities among *many* systems.

This domain analysis task is made more complex by the fact that information comes from multiple sources such as source code, requirements and design documents, domain literature, and domain experts. Assimilating all this information presents a difficulty in information management. The requirement that this information and other end-products be general in form and in a format that can be exploited by others complicates the requirements for representation mechanisms.

In order to facilitate the evaluation of existing domain analysis methods and their products, a set of classification and evaluation criteria is very helpful [Firth 87]. Criteria for classifying and evaluating the existing products and processes of domain analysis can assist the development of a domain analysis method. A set of appropriate criteria can be broken down into three major areas:

1. The *process* aspect considers: how the domain analysis method will affect an organization (e.g., evolutionary vs. revolutionary); how to manage and maintain the products; how the producer gathers, organizes, validates, and maintains information; and how the users can effectively apply the products in the development.
2. The *product* aspect considers: the types of products that are generated by the method; how they are represented; and how applicable they are in applications development.
3. The *tool support* aspect considers the availability of tools and the extent to which the tools support the method. It also looks at how well the support tools are integrated, their ease of use, and their robustness.

For the purposes of this study, emphasis will be placed on process and product considerations of the different methods. The level of tool support is less important than the method embodied in the tool.

2.2. Comparative Study

A comparative study of domain analysis techniques will provide a better understanding of the range of approaches. Several of the studies listed in the previous section describe differing approaches to the techniques of domain analysis. The following four approaches will be the focus of this section:

1. Genesis/University of Texas: a tool for constructing database management systems (DBMSs). An analysis of the DBMS domain is the basis of a generic architecture, components, and the constructor.
2. KAPTUR (Knowledge Acquisition for the Preservation of Tradeoffs and Underlying Rationales)/CTA: a tool to facilitate knowledge representation and analysis. A domain analysis provides a baseline structure; any deviations from the baseline are stored as *distinctive features* of the domain, along with a rationale for the new features.
3. DESIRE (DESIgn REcovery)/MCC: a tool that supports recovering and reengineering a design from existing code. This is one of the parts of performing a domain analysis.
4. SPS (Software Productivity Solutions): guidelines for conducting a domain analysis.

The following sections describe each of these methods and contrast it with the FODA method.

2.2.1. The Genesis System

Genesis is a tool system developed by Don Batory of the University of Texas ([Batory 88b], [Batory 88c]). The goal of this effort was to develop a "database compiler" to synthesize customized DBMSs from pre-written components. The developers report that:

> Enormous increases in software productivity are achieved by exploiting reusable and plug-compatible modules. The popularized, but mythical, concept of 'software ICs' is actually a reasonably accurate description of our technology [Batory 88c].

The system has been under development for ten years and can now generate centralized relational DBMSs with various configurations. Current efforts include extending the prototype to support object-oriented and distributed databases.

The developers used domain analysis techniques to formulate a building-block technology for file management systems. Their approach, similar to the FODA method, uses "existing systems, published algorithms, and structures to discern *generic architectures* for large classes of systems." [Batory 88c] The Genesis method also defines the generic architecture as the primary product of the domain analysis. While the method recognizes the need to gain a full understanding of a domain in order to construct the architecture, Genesis does not give specific techniques for obtaining or representing this understanding. The developers of Genesis point out, however, that their domain model has been evolving for years.

The Genesis generic architecture is a template for constructing systems. Building-block

modules with standardized interfaces can be "plugged" into this template by the Genesis tool at the direction of the user. The Genesis approach recommends studying domain information, including systems, algorithms, and data structures, to produce an architecture and interfaces that cover the class of systems. This product forms a blueprint for constructing software systems from building blocks.

The Genesis approach describes the two steps to the analysis process:

1. Defining a generic architecture: a software architecture is defined in terms of a hierarchy of "independently-definable objects" (IDOs): objects whose implementations have no impact on other objects. For each IDO, there are operations, standard interfaces, and a collection of algorithms that implement the operations. The method does not produce a domain model, domain terminology dictionary, or method of mapping the DBMS problem domain to the IDOs, as would be required by the FODA method.
2. Defining standardized interfaces to objects: in addition to the architecture information, the Genesis method also produces information about the interrelationships between the building-blocks (i.e., IDOs, operations, and algorithms). When the tool is executed, these constraint rules eliminate all building blocks that are incompatible with others already selected by the user. These constraints are similar to rules for combining required, optional, or alternative features in the FODA method.

Genesis offers some guidelines for creating a software architecture, but in general the method relies on the knowledge, experience, and creativity of domain experts to create the architecture. The underlying problem domain is not represented. In addition, the method does not capture information to support requirement or design decisions. A developer using the Genesis tool must be an expert in the domain and should know what to select and the implications of the selection. To improve its utility, the tool should provide users an interactive design aid that explains the effects of each design decision.

The Genesis developers have recognized many of the issues that have also been used in creating the FODA method.

- Establishing the proper decomposition of the domain is essential. If the subdomains are too restricted, the analysis may lead to interfaces that are unstable, with changes required each time new algorithms or structures are introduced. If the decomposition is too broad, the grouping of algorithms and structures will link objects that really should be separate.
- Developing the architecture is not simply listing algorithms and structures and allowing a developer to select what appears to support his requirements. The domain analysis must provide an architecture for building a system from these orthogonal units (IDOs).
- The technology embodied in Genesis is intended to support the exploration of solutions to problems in the domain. The architecture provides "a platform on which to implement a class of previously identified solutions." [Batory 88c]

The developers of Genesis have validated a reuse approach based on domain analysis.

Their method is built upon identifying the underlying IDOs of a domain, fashioning a generic architecture for structuring the IDOs, constructing the building blocks to implement these structures and implementing the tool to perform system synthesis. The FODA method is directed towards generalizing that approach for other domains.

2.2.2. MCC Work

The Microelectronics and Computer Technology Corporation (MCC) has developed domain analysis methods and related tools to support software reuse ([Biggerstaff 89a, Biggerstaff 89b, Lubars 87, Lubars 88]). These tools include:

DESIRE: A tool for recovering design from code and re-engineering the recovered design.

ROSE: A tool for representing design templates and generating code from the templates.

TAO: An expert tool for domain understanding and modeling (in concept development stage).

Each of these tools was developed independently of the other, leading to an integration problem. For instance, the designs recovered by DESIRE cannot be accessed by ROSE. In addition, there is no single underlying domain analysis approach; DESIRE is built upon reverse engineering concepts, while ROSE selects abstract designs to meet requirements and specifications supplied by a user.

2.2.2.1. The DESIRE Design Recovery Tool

The MCC *DESIgn REcovery* method gathers design information from source code, organizes the information, and abstracts a design. The DESIRE tool performs these steps and allows domain experts to display and restructure the design, incorporating their knowledge. The recovery results have been positive, and there are active users of the tool that can process C and C++ programs of moderate size (50 - 100,000 lines of code).

The main product of this method is a generic design which MCC calls a *domain model*. Various aspects of this model, such as the calling structure and module structure, can be captured and displayed using the tool. However, there is no way to abstract the design model into an architecture model (e.g., a layered architecture that shows packaging of routines) or to access design decision information. The MCC design recovery method also does not provide any model of the problem space. Both design criteria and problem space modelling are important aspects of the FODA domain analysis method.

DESIRE is used primarily as a tool for maintaining or re-engineering existing code. To truly support domain analysis and reuse, the design recovery method and tool must provide knowledge to support an analysis of the features of the problem domain. One could then use the tool to explore the extent to which existing systems exhibit these features, and extract their implementation. The information extracted from the code could be structured to represent the implementation in a format (such as a design template) that could be reused in other developments.

2.2.2.2. Domain Analysis Method

The MCC domain analysis method is primarily embodied in the Reuse of Software Elements (ROSE) system. This tool attempts to capture domain knowledge to support early reuse decisions. The tool is aimed at meeting four goals:

1. Feature based selection: identifying domain objects and associated components from their known features.
2. Constraint based analysis: analyzing a set of user requirements based on known domain dependencies and relationships.
3. Domain-driven completion: performing completeness checks to ensure there are no missing requirements in a system specification.
4. Domain-driven refinement: collecting and applying design issues as the criteria for support of design decisions.

The goals of the MCC domain analysis method directly match those of the FODA method. However, while stating these goals, the MCC approach does not provide specific techniques for capturing or representing this information. The ROSE tool embodies some of these elements and has been used experimentally to produce small programs (up to 600 lines of code). ROSE assumes that the domain information already exists, and does not support its creation. The tool remains a prototype for exploring the concepts of design analysis, but is useful in defining requirements for domain analysis support.

2.2.3. CTA Work

Computer Technology Associates (CTA) has developed a domain analysis method and a series of prototype support tools. CTA has used its method, independent of the tools, to perform a domain analysis of NASA payload operations control center (POCC) software [Moore 89]. This effort was a major factor in setting requirements for a tool currently under development called KAPTUR (Knowledge Acquisition for the Preservation of Tradeoffs and Underlying Rationales) [Bailin 89]. KAPTUR will embody CTA's current approach to domain analysis.

One of the major concepts behind KAPTUR is the ability to apply the tool to successive system developments in the same domain and have KAPTUR semi-automatically support the detection of significant differences between these systems. These differences are referred to in KAPTUR terminology as *distinctive features*. Distinctive features consist of (1) new functions, (2) new subsystems, or (3) new placements of existing subsystems.[3] As new systems are input into KAPTUR, the user must identify and name new capabilities, or use the same name for a feature that is identical to one present in a previous system. Given this information, KAPTUR can automatically build a set of knowledge base relationships that connect the different systems sharing common features. (This is done using a Prolog program.)

[3]Note that these features are not decomposable into lower-level features as such, although the function, subsystem, etc. which constitutes the distinctive feature may be decomposed like other functions, subsystems, etc. in KAPTUR's knowledge base description.

The CTA approach highlights the roles played by domain developers ("supply side") and by system developers ("demand side") to define a reuse-based life-cycle, much as the FODA method establishes the domain analyst as supplying information to the user community. (Figure 2-1 shows the CTA roles and equivalent FODA method roles in italics under each CTA label.) In the CTA approach, the "develop-*for*-reuse" expert establishes the scope of the domain knowledge that is commonly used across the systems in the domain. The domain developer organizes this knowledge in the domain library in a consumable form. The "develop-*by*-reuse" expert uses the domain library to discover resources available for development and provides them to the system developers who apply these resources to implement systems. The system developers feedback new reuse requirements to the domain developers to improve the domain library. This process very closely matches that of the FODA method.

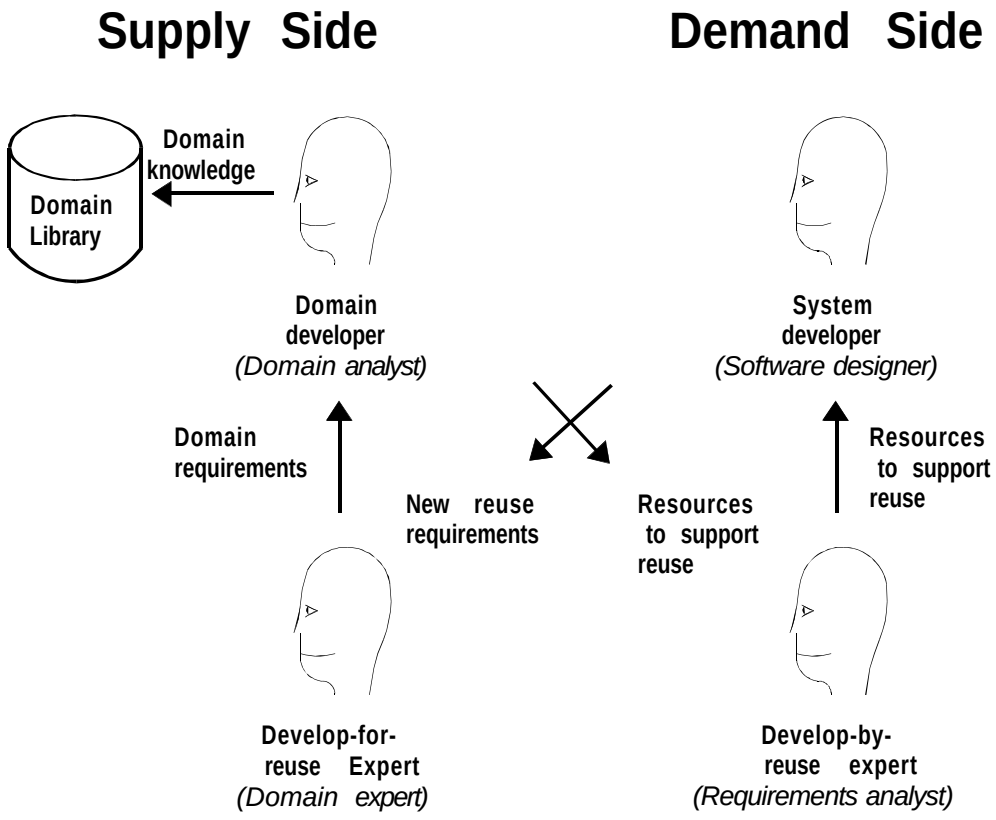


Figure 2-1: Participants in CTA's Reuse-Based Development Process

The CTA method produces many of the same products considered essential by the FODA method. The CTA domain model is stored as a hypertext network to represent system

structure. The network includes system features such as subsystems, functions, resources (documents or people), external entities, and parameters. In addition, the links carry relational information, such as composition/decomposition, inputs/outputs, interactions, etc. Because the model is intended to capture the domain knowledge of system developers, the network also includes design decision information. This includes justifications that represent design issues, positions, and arguments, as well as explanations to help users understand the system. The method also advocates the use of entity-relationship and data flow diagrams as conventional specification methods.

CTA is developing the KAPTUR tool to support its method. One of the salient features of the KAPTUR tool is its ability to store and trace the decision process that leads to a specific architecture. This reasoning process is captured in the database, an idea previously embodied in the gIBIS (graphical Issue Based Information Systems) [Begeman 88] system from MCC at a greater level of detail. FODA also makes use of design decision information, linking it to specific features of the domain.

2.2.4. SPS Work

Software Productivity Solutions (SPS) has developed guidelines for performing a domain analysis [Gilroy 89]. SPS defines domain analysis as: "the systems engineering of a family of systems in an application domain through development and application of a reuse library" [Gilroy 89]. The process proposed in their guidelines follows the phases of the FODA method. The SPS method defines the goal of domain analysis as the production of a "reuse library asset that will be used in the implementation of system instances." Although the method includes the domain model and generic architecture as assets, the method does not provide concrete examples of these products.

The SPS method proposes a three-step process for domain analysis:

1. *Model the domain:* to scope the domain and develop "a complete specification of the domain, accomplishing a sort of requirements analysis."
2. *Architect the domain:* to develop and validate a generic, object-oriented Ada architecture for the family of systems.
3. *Develop software component assets:* to build a set of reusable object-oriented Ada components and catalog the components into a library for the domain.

The FODA method is similar to the SPS method in that both include separate scoping, modelling, and architecture phases. The SPS method differs in its emphasis on Ada architectures and components as the products of the analysis.

A detailed view of the SPS method also reveals similarities to the FODA method. The scoping activity in the SPS method produces a "domain planning document" to guide the analysis. The other modelling activities produce a domain model based on the planning document and knowledge of previous systems. While the FODA method concentrates on user-visible features, the SPS method seems to concentrate on internal characteristics of the identified objects, such as data flow and state transition. FODA also includes this internal

view in the functional descriptions, but it is not the primary focus of the method. This domain modelling phase also generates a domain dictionary, similar to FODA's domain terminology dictionary.

The SPS domain architecture provides design information for building Ada software components. This view of an architecture contrasts with that of the FODA method. Both SPS and FODA provide dynamic/static requirements information as part of the domain model. SPS then turns this information into Ada package specifications for reusable code. The FODA method relies on a generic set of parallel processes to define the control aspect of the architecture, and allocates specific functions defined in the domain model to modules that these processes control. The available documentation for the SPS method also does not provide any example of its architectural methods.

Although the SPS method lacks detailed guidelines for representing the products of a domain analysis, it is useful in establishing a process for domain analysis. The method is also valuable in connecting domain analysis to the development process as a whole. The SPS process model served as an example in defining the FODA methods.

3. Overview of the Feature-Oriented Domain Analysis (FODA) Method

The FODA method supports reuse at the functional and architectural levels. Domain products, representing the common functionality and architecture of applications in a domain, are produced from domain analysis. Specific applications in the domain may be developed as *refinements* of the domain products.

Domain analysis is related to requirements analysis and high-level design, but is performed in a much broader scope and generates different results. It encompasses a *family* of systems in a domain, rather than a single system, producing a domain model with parameterization to accommodate the differences and a standard architecture for developing software components. An ideal domain model and architecture would be applicable throughout the life cycle from requirements analysis through maintenance.

3.1. Method Concepts

The development of domain products that are generic and widely applicable within a domain is a primary goal of the FODA method. This genericness (i.e., general knowledge) can be achieved by abstracting away "factors" that make one application different from other applications. However, to develop applications from the generic products, those factors that have been abstracted away must be made specific and reintroduced during refinement. Not only the genericness, but also those factors that make each application unique are an important part of the domain knowledge and should be captured in the domain products. With this method, domain products are not ends unto themselves, but evolve though applications.

The underlying concepts of this method are discussed in the remainder of this section. Section 3.2 describes some of the sources of domain information. A summary of the domain analysis activities and products is included in Section 3.3.

3.1.1. Modelling Primitives

The maturity of an engineering field can perhaps be indicated by the level of standardization of the design of products in the field. No cars are designed from scratch these days: design frameworks have been standardized over time, and new features are added to an existing design framework to develop a new model. Software development, like other engineering fields, can benefit from the development and reuse of "product frameworks" in an application domain (i.e., a product line or a product family). The "product frameworks" in the context of software are *abstractions* of functionalities and designs (i.e., architecture) of the applications in an application domain.

To support the development and reuse of "abstractions," this method is founded on a set of modelling concepts. They are:

- aggregation/decomposition

- generalization/specialization [Borgida 84]
- parameterization [Goguen 84]

Abstracting a collection of units into a new unit is called *aggregation*. For example, school is an aggregation of students, teachers, etc. Refining an aggregation into its constituent units is called *decomposition*. This modelling primitive allows composition of components into a new aggregated component or decomposition of an abstract component into its parts. The stepwise refinement method [Wirth 71] is based on this aggregation/decomposition concept.

Abstracting the commonalities among a collection of units into a new conceptual unit suppressing detailed differences is called *generalization*. Refining a generalized unit into a unit incorporating details is called *specialization*. For example, the conceptual entity "employee" is an abstraction of secretaries, managers, technical staffs, etc. This modelling primitive allows the development of generic components that can be refined in many different ways.

Parameterization is a component development technique in which components are adapted in many different ways by substituting the values of parameters which are embedded in the component. It allows codification of how adaptation is made within a component. The Ada generic is one example of parameterization.

The FODA method applies the aggregation and generalization primitives to capture the commonalities of the applications in the domain in terms of abstractions. Differences between applications are captured in the refinements. An abstraction can usually be refined (i.e., decomposed or specialized) in many different ways depending on the contexts in which refinements are made. Parameters are defined to uniquely specify the context for each specific refinement. The result of this approach is a domain product consisting of a collection of abstractions and a series of refinements of each abstraction with parameterization. Whenever a new refinement (i.e., a refinement that is not already included in the domain product) is made, the context in which the refinement is made must be defined in terms of parameters before it is incorporated into the domain product. Therefore, in the FODA method, a domain product is not the end-product; it expands and evolves through application.

Understanding what differentiates applications in a domain is most critical in domain analysis since it is the basis for abstraction, refinement, and parameterization. The FODA method focuses on identifying factors that can cause differences among applications in a domain (both at the functional and the architectural level) and uses those to parameterize domain products. The concept of parameterization and the types of factors considered in this method for parameterization are described in the following section.

3.1.2. Product Parameterization

The purpose of parameterization is to develop generic components that can be adapted in many ways by supplying values to parameters. This is not a new software engineering concept; it has been implemented in many forms such as subroutines, generics, macros, and preprocessors. However, these techniques have mostly been applied to code, although code is not the only software engineering product. The FODA method applies this concept to other software engineering products including requirements and design.

Identifying and understanding "factors" that result in different applications or different implementations of similar applications in a domain is the basis for parameterization. The factors considered in this method for parameterization can be largely classified into commonalities and differences of:

- The *capabilities* of applications in a domain from the end-user's perspective.
- The *operating environments* in which applications are used and operated.
- The *application domain technology* (e.g., navigation methods in the avionics domain) based on which requirements decisions are made.
- The *implementation techniques* (e.g., synchronization mechanisms used in the design).

Applications in a domain, although they provide a large set of common capabilities, provide different sets of capabilities, which make each application different from others. These capabilities from the perspective of end-users are modelled as *features*. Some examples of features from the window manager domain are *tiled* and *overlapped* window layouts, and *listener* and *real estate* window selection modes. A set of features (e.g., an *overlapped* window system with *listener* mode) from the window manager domain characterizes the capability of a window manager. Therefore, features are used to parameterize domain products from the user's perspective.

It is important to understand the use of the term *user* in this context. In Section 1.2 a user is defined as being either a person *or* an application that operates a system. In this sense the term *user* can include the interactive user of a database management system (DBMS), the *developer* of an inventory application built on top of the DBMS, or the inventory application *itself*. All of these "users" require similar, but slightly varying knowledge of the interfaces to the DBMS and how to operate it. In a practical sense the scope of the term "user" should be taken to apply to the "interface" which is of most relevance; the interactive interface, the application programmer interface (API), or other interfaces.

Applications may run in different operating environments. They may run on different hardware or operating systems, or interface with different types of devices. Understanding the commonalities and differences between the external components with which the applications interface is essential to abstracting the functionalities of those external components and defining common interfaces to them.

There are many ways to provide the features desired by the end-users. Different people involved in the development can make different decisions, all of which affect how the features are implemented. Requirements analysts select a set of domain technologies and define internal functions based on the selected technologies. For example, the choice between *pixel-based* and *paint-and-stencil* imaging models determines what internal functions the display software should provide.

Designers also see many ways to implement the internal functions defined by requirements analysts. Considering the hardware and software platforms, the space and time constraints, the expected transaction volume and frequency, etc., designers make a number of design

decisions which all affect the formulation of software products. For example, a decision as to whether a window system will be implemented based on the *kernel-based model* or the *client-server* model affects the structure of the system. Various technical factors, from the perspectives of requirements analysts and designers, that can result in different product structures (e.g., different functional decompositions or design structure) are defined as *issues* and *decisions*. The *issues* and *decisions*, along with *features*, are used to parameterize the functional and the architecture models of the domain.

In the development of an application, earlier decisions generally affect the range of decisions that can be made later, as shown in Figure 3-1. For instance, any prior decisions on an operating environment may affect the range of architecture models and implementation techniques that can be selected later. In the window management domain, for example, a *kernel-based* window system might be appropriate in a non-multitasking environment. The knowledge of inter-relationships between various development decisions is an important part of domain knowledge.

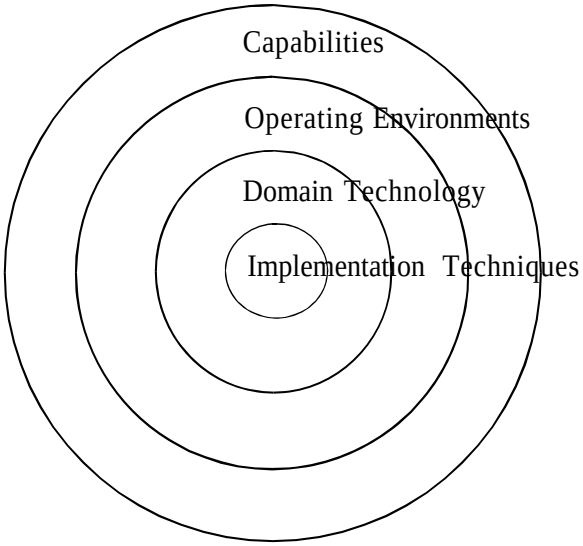


Figure 3-1: Types of Development Decisions

3.1.3. Levels of Abstraction

To maximize reusability, the FODA method advocates that applications in the domain be abstracted to the level where differences between the applications are not revealed. This abstraction is accomplished by using the modelling primitives discussed in Section 3.1.1. The factors that make each application unique are subsequently incorporated into the refinements of the abstractions, but incorporation should be delayed as much as possible. For example, "x" in Figure 3-2 represents the commonalities of all applications considered in the domain analysis. As "x" is refined, the information on the context in which each refinement is made is incorporated into the refinement. Therefore, "x1" and "x2" in Figure 3-2 are more context sensitive and less reusable than "x."

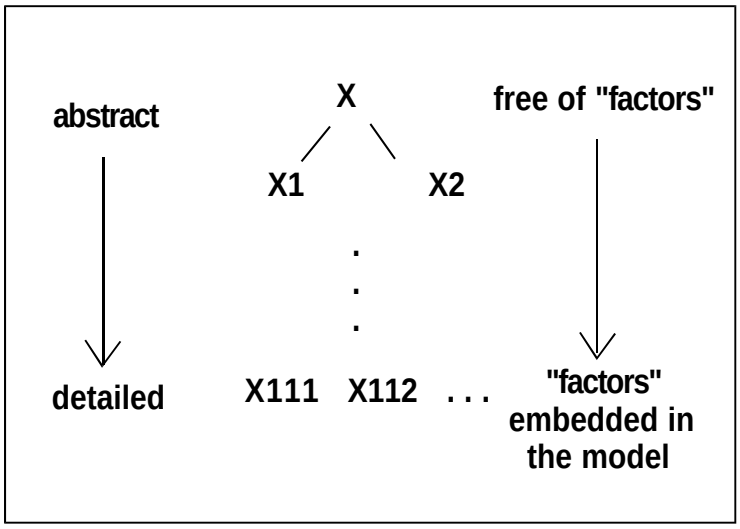


Figure 3-2: Model Abstraction

A generic component that captures only the common characteristics of applications in a domain and, therefore, is free of those "factors" discussed in Section 3.1.2 has a high degree of reusability because of generality. However, the "productivity increase"[4] from the reuse of a generic component may not be as high as the productivity increase from the reuse of a component that is less generic but addresses specific features of the intended application, as shown in Figure 3-3. The FODA method gives the benefits of both approaches by providing generic components and also refinements of those components at various levels. As generic components are refined the factors that make applications unique are incorporated into the refinements. Reuse of the components can happen at the level that is most appropriate for an application.

Domain products (i.e., abstractions and refinements) embody a wide range of domain knowledge. During the domain analysis, the knowledge is collected from various sources and then organized and represented as domain products. Some of the important sources of domain knowledge are discussed in the next section.

3.2. Information Sources

There are several information sources that may be available for a domain analysis, each with distinct advantages and drawbacks in the types of information it offers. Table 3-1 provides an overview of information sources to use while gathering information for the domain analysis.

[4]The phrase "productivity increase" here refers to the reduction in total software development effort from applying the component.

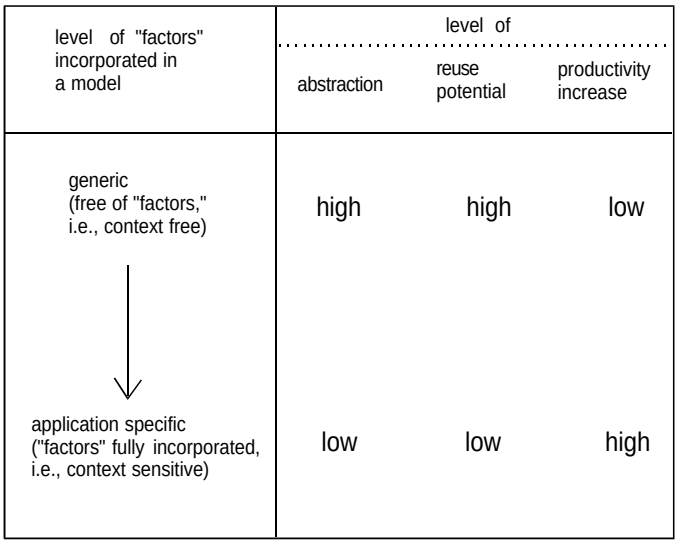


Figure 3-3: Model Reuse

Source	Advantages	Disadvantages
Textbooks	• Good source of domain knowledge, theories, methods, techniques, models	• Reflects only specific author's views • May use idealized or biased models
Standards	• Represents standard reference model for domain	• Model may not be current with new technology
Existing Applications	• Most important source of domain knowledge • Can be directly used to determine user-visible features • Requirements documents available for domain model • Detailed design & source code show architectures	• Cost of analyzing many systems is high
Domain Experts	• Can provide contextual/rationale information unavailable elsewhere • Can be consultant during DA, validator of products afterwards	• Experts have different areas of expertise; several experts may be needed

Table 3-1: Domain Analysis Information Sources

As an example of the use of textbooks, in the database domain there are many books describing the relational theory [Codd 70] that is the basis of many database management systems. An understanding of the theory is essential to understand the database management systems. Also, many textbooks describe the principles of different domain-specific techniques, such as hashing and indexing methods.

In the use of existing applications it should be pointed out that systems should be selected that are as divergent (or "orthogonal") in functionality as possible, while still being in the domain. This helps to minimize the number of systems that must be examined. As a general guideline, at the very least three systems should be used as inputs to the domain analysis.

Domain products are produced through a number of activities. The domain analysis activities of this method and their inter-relationships are discussed in the next section.

3.3. Activities and Products

The domain analysis process consists of a number of activities and many models are produced from the process. Table 3-2 summarizes a grouping of the activities into phases, with the inputs to and outputs from each activity, and shows how the outputs from one activity are used in other activities.

The models produced from a domain analysis are used to develop applications in the domain, as depicted in Figure 3-4. The context model can be used by a requirements analyst to determine if the application required by the user is within the domain for which a set of domain products is available. If the application is within the domain, then the feature model can be used by the requirements analysts to negotiate the capabilities of the application with the users. The feature model (described in detail in Section 5.1.2) identifies common (i.e., *mandatory*), alternative, and optional features. Where the terms relevant to the features are new or unclear, the domain terminology dictionary contains descriptions of their meanings. Typically, a data-flow model has been used as a communication medium between users and developers. However, a data-flow model contains definitions of the internal functions and does *not* contain the information that the user needs most, which is a description of the *external,* or *user-visible* aspects of the system. The feature model is a better communication medium since it provides this external view that the user can understand.

The entity-relationship model can be used by a requirements analyst to acquire knowledge about the entities in the domain and their inter-relationships. An understanding of the domain will help the analyst to deal with the user's problems. The analysis can determine if the functional model, consisting of the data-flow model and the finite state machine model of the domain products, can be applied to the user's problems to define the requirements of the application. If the user's problems are all reflected in the feature model, then the requirements may be easily derived from the models by tracing the features embedded in the models as parameters. Otherwise, new refinements of the abstract components may have to be made. The architecture model is used by the designer as a high-level design for the application. Again, if the user's problems are reflected in the feature model, a design may be easily derived from the architecture model. If the problems are not represented, then the architecture model should be further refined from the other domain products.

Detailed discussions of the domain analysis activities and models are included in Chapters 4 through 6.

Table 3-2: A Summary of the FODA Method

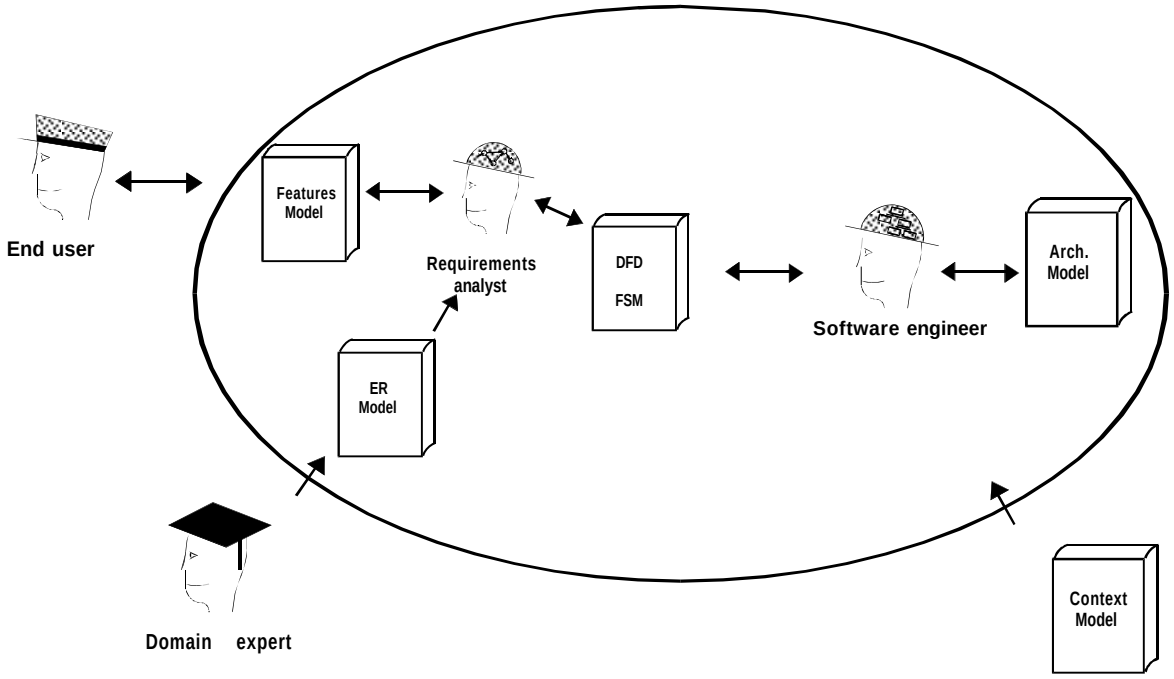


Figure 3-4: Use of Domain Analysis Products in Software Development

4. FODA Context Analysis

4.1. Purpose

The purpose of context analysis is to define the scope of a domain that is likely to yield exploitable domain products. In the context analysis phase the relationships between the candidate domain and the elements external to the domain are analyzed, and variability of the relationships and the external conditions (e.g., different applications using the domain and their data requirements, different operating environments, etc.) are evaluated. The results of the context analysis, along with other factors such as availability of domain expertise, domain data, and project constraints, are used to scope the domain.

The final results of the context analysis are documented in a context model. This context model defines the boundary of the domain, that is, the scope of the domain modelling which follows the context analysis.

4.2. Model Description

A context model consists of one or more structure diagrams and data-flow diagrams. A structure diagram is an informal block diagram in which the target domain is placed relative to higher, lower, and peer-level domains. The higher level domains, if any, are those of which the target domain is a part. If there is no higher level domain, then the types of applications in the domain should be identified. The lower level domains (i.e., sub-domains) are those that are within the scope of the target domain, but are well understood. Previous domain analysis results or standards are available and, therefore, they will not be included in the analysis. Any other domains that interface with the target domain (i.e., peer-level domains) must also be in the diagram. More than one structure diagram may be used if necessary. (An example structure diagram may be found in Figure 7-3 on page 60.)

A data-flow diagram shows data-flows between the target domain and all other domains and entities with which the target domain communicates. (See Figure 7-4 on page 61 for an example.) One thing that differentiates the use of data-flow diagrams in domain analysis from other typical uses is that the variability of the data-flows across the domain boundary must be indicated in the diagram. This may be done with a set of diagrams, each describing a different context, or with one diagram with the text describing the differences. If the variations are due to different features of the applications in the domain, the variations must be described in terms of the features. (Context modelling and features modelling may have to be performed in parallel or iteratively until the context model becomes complete, at which time detailed feature modelling may start.) Textual descriptions of the functionality and reusability of the target domain, the objectives of the domain analysis, and any standards that are relevant must also be included in the context model.

The sub-domains that need not be included in the analysis are also identified in the diagram.

These are the domains for which the results of previous domain analyses exist as standards, reusable components, and/or domain models. These sub-domains should be identified as lower-level domains in the structure diagram and as external entities in the data flow diagrams. Standards or other domain analyses results that apply to the external entities should be identified in the diagram.

For each entity in the context model, the following information should be included in the document.

- Name of the entity (an object on the diagram).
- Description of the function for a functional entity or description of the contents for a data entity.
- Applicable standards and/or reusable components.
- Description of variability, including the range, frequency, and binding time (i.e., compile-time, activation-time, and runtime) of the variation (see Section 5.1.2 for details of feature binding times).
- Other items describing the attributes of the entity.
- Source for the information or for additional information.

Other information which would be appropriate to include in the context model would be the applicable or related features from the feature model produced during the domain modelling phase (see Section 5.1). The incorporation of this information into the context model implies either an "iterative" or a parallel approach to the context analysis and domain modelling phases, since in a strictly sequential approach the feature model would not be available to the context analysis (it is developed in the next phase of the analysis). The feature model information may be useful in determining the scope of the analysis.

4.3. Model Usage

One of the necessary conditions for building reusable software is an understanding of the different contexts in which the reusable software is to be incorporated or operated. This understanding of the extent of contextual differences, and when and how frequently the context changes, will help software developers decide if software that meets the requirements can be built and, if so, what to parameterize and how to structure software so that it can be adapted to different contexts as needed. For example, in an environment where there are many different types of terminals, the user interface part of the software should be built so that it can handle different terminals without modification.[5] In order to be able to build common user interface software, the commonality and differences of terminals (to abstract and define common interfaces) and their usages (to build common utilities) must be understood.

The understanding of contextual variations will help rescope the domain. If there is a high degree of variation in context and the contextual differences cannot be abstracted away, it

[5]Precisely this capability is provided by the UNIX "curses" and "termcap" facilities.

might indicate either that there is no commonality, or that the domain needs to be rescoped to a narrower domain.

The context model defines the scope of a domain analysis. Other subsequent analysis activities, such as feature modeling, functional modelling, entity-relationship modelling, and architecture modelling, are all performed within the scope defined in the context diagram. The data-flow context diagram is used as the starting point in the functional modelling.

4.4. Process and Guidelines

To yield domain products that can be exploited in other applications, the scope of a domain should be decided considering: (1) the commonality of the domain in existing systems, (2) the availability of information on the domain and domain expertise, (3) the expected usages of the domain products, and (4) the project resources and constraints. Keeping these factors in mind, the activities for context analysis and domain scoping are identified as follows:

1. Make an initial "cut" of the target domain and the domain boundary. (It is assumed that a candidate domain for context analysis is already selected.) Identify the existing applications in the domain or applications using the domain. Identify information sources and any previous works on the domain including domain analysis products, standards, and books and technical papers.
2. For each application identified for the analysis, describe the context of the candidate domain in terms of structure diagrams and data-flow diagrams. Verify that the domain has a clear physical boundary within the application. If there are any previous domain analysis results or standards, determine if they address each application adequately; record problems, if any.
3. Understand the usage of the target domain, any recent technical developments that will affect the domain, and any future plans or requirements.
4. Analyze the variability of the usages and the contexts of each use.
    a. Based on the features of applications in the domain, begin the definition of a feature model (See Section 5.1 on the feature model).[6]
    b. Identify the environmental differences (i.e., operating environments).
    c. Identify any assumptions (sub-domains and standards) on which the target domain is based.
    d. Construct a common model by classifying specifics of the contexts into general categories so that each context can be defined as an instantiation of the common model.
5. Evaluate the model against the applications used in the analysis and incorporate the variability information (i.e., how the common model can be adapted to each context) into the context model.

[6]Again, use of the feature model in the context analysis phase requires iterative or parallel development of the context and domain models.

6. Evaluate the commonality of the applications, feasibility of constructing domain products, and potential uses and benefits of the domain products.
7. Estimate the resources for the subsequent activities considering availability of domain experts, previous work, and maturity of the domain.
8. Document the context model; define the terms used in the model in the domain terminology dictionary.
9. Validate the model against at least one application that is *not* included in the analysis. Also, have the model reviewed by domain experts. Record the validation results in the context model documentation.

Within the scope defined from the context analysis, the problems pertaining to the domain are analyzed and modelled in the domain modelling phase. Chapter 5 describes domain modelling activities.

5. FODA Domain Modelling

For the domain that is scoped in the context analysis, commonalities and differences of the problems that are addressed by the applications in the domain are analyzed in the domain modelling phase and a number of models representing different aspects of the problems are produced. The domain modelling phase consists of three major activities: feature analysis, entity-relationship modelling, and functional analysis. Each of these activities is described in the following sections.

5.1. Feature Analysis

5.1.1. Purpose

The purpose of feature analysis is to capture in a model the end-user's (and customer's) understanding of the general capabilities of applications in a domain. Typically, data-flow diagrams have been used by software developers as a medium to communicate requirements with customers (and also with designers). However, data-flow diagrams include definitions of software *functions* for satisfying customers' needs and, often, such information is not of interest to customers. Customers need to know the essential capabilities of the application that satisfy their needs. These capabilities might include features such as:

1. services provided by the application,
2. performance of the application,
3. hardware platform required by the application,
4. cost,
5. others

Our approach to analysis focuses on an end-user perspective of the functionality of applications, that is, the "services" provided by the applications and the operating environments in which the applications run.

Since the primary interest is in the commonality of a family of applications, the feature model should capture the common features and differences of the applications in the domain. The features in the feature model will be used to generalize and parameterize other models.

5.1.2. Model Description

Features are the attributes of a system that *directly affect* end-users. The end-users have to make decisions regarding the availability of features in the system, and they have to understand the meaning of the features in order to use the system. Examples of such features are the *call-forwarding* and *call-transfer* features of a telephone switching system, and the *automatic* and *manual* transmission features of an automobile, as illustrated in Figure 5-1. When a person buys an automobile a decision must be made about which transmission

feature the car will have, as it is not possible to have both.[7]

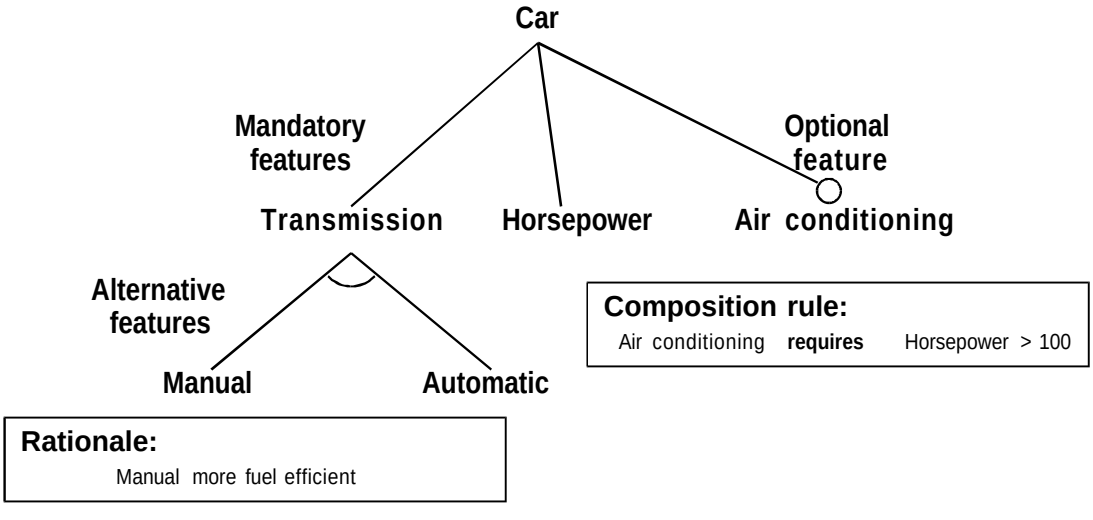


Figure 5-1: Example Showing Features of a Car

A feature model represents the *standard* features of a family of systems in the domain and relationships between them. The structural relationship *consists of*, which represents a logical grouping of features, is of interest. *Alternative* or *optional* features of each grouping must be indicated in the feature model. For example, the *automatic* and *manual* features are alternatives and the *air-conditioning* feature is optional, as indicated in Figure 5-1 by small arcs and circles, respectively. Each feature must be named distinctively and the definition should be included in the domain terminology dictionary.

Alternative features can be thought of as specializations of a more general category. For example, the automatic and manual transmission features can be thought of as specializations of the general "transmission" feature. The term "alternative features" is used (rather than "specialization features") to indicate that no more than one specialization can be made for a system. However, the attributes of (i.e., the description made for) a general feature are inherited by all its specializations.

Composition rules define the semantics existing between features that are not expressed in the feature diagram. All optional and alternative features that *cannot* be selected when the named feature is selected must be stated using the *"mutually exclusive with"* statement.[8] All optional and alternative features that *must* be selected when the named feature is selected must be defined using the *"requires"* statement.

---

[7]At the time of the publication of this report a car was announced which had both types of transmission available simultaneously. This is a common evolution of features over time and is discussed in Section 8.2.1.

[8]The syntactical form and automated tool used to process this statement is presented in Section 7.3.2.6.

Selection of optional or alternative features is not made arbitrarily. It is usually made based on a number of objectives or concerns that the end-user (and customer) has. For example, in the case of buying an automobile, if the buyer's only concern is fuel efficiency, then he might want to take the manual transmission feature and the smallest available engine, perhaps a diesel if that option is available. However, if the user's concern is fuel efficiency and maintenance cost, and if the maintenance cost of a diesel engine were much higher than that of a gasoline engine, then he might want to take the gasoline engine option instead, even if the fuel efficiency is not as good as that of a diesel engine. A set of issues and alternative decisions in the selection of optional and alternative features is captured using the form in Appendix A.5.

One of the fundamental trade-offs a system architect makes is deciding when to "bind" or fix the value of a feature, as this will have an impact on the final architecture. For the purpose of generalization and parameterization of the software architecture, alternative and optional features are grouped into three classes based on when the binding of those features (i.e., instantiation of software) is done, as depicted in Figure 5-2.

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

Load-time features: features that are selected or defined at the beginning of execution but remain stable during the execution. Examples of this class of features are the features related to the operating environment (e.g., terminal types), and mission parameters of weapon systems. Software is generalized (e.g., table-driven software) for these features, and instantiation is done by providing values at the start of each execution.

Runtime features: features that can be changed interactively or automatically during execution. Menu-driven software is an example of implementing runtime features.

Documentation of a feature model includes: a structure diagram showing a hierarchical decomposition of features indicating optional and alternative features, definition of features, and composition rules of the features. Each feature definition should include the information stated in the sample form in Appendix A.1.

5.1.3. Model Usage

The feature model serves as a communication medium between users and developers. To the users, the feature model shows what the standard features are, what other features they can choose, and when they can choose them. To the developers, the feature model indicates what needs to be parameterized in the other models and the software architecture, and how the parameterization should be done.

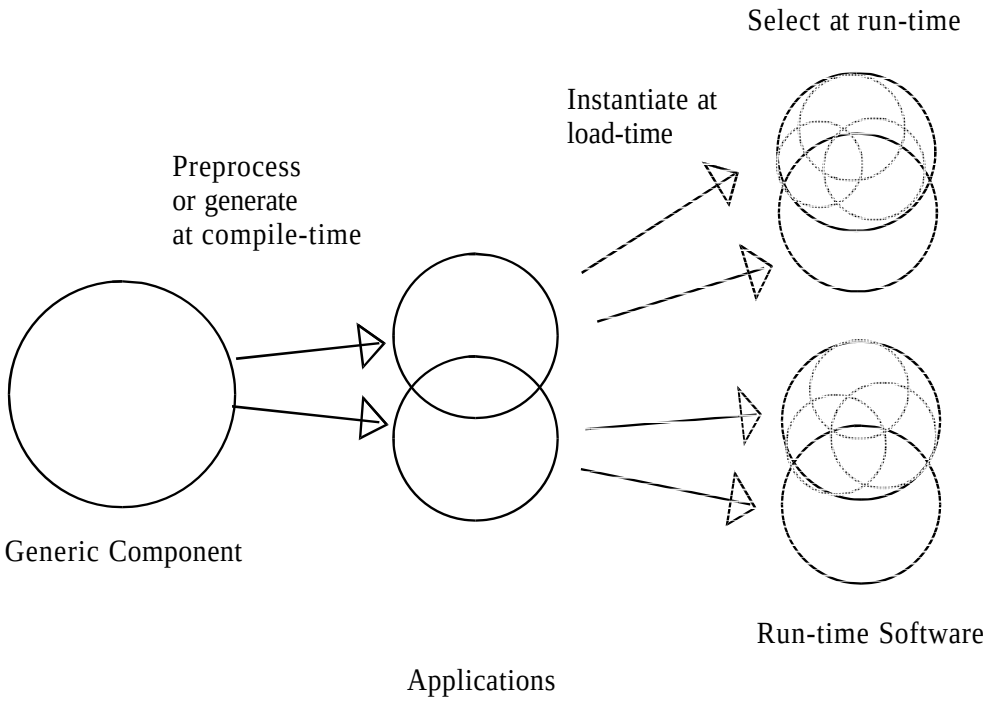


Figure 5-2: Processing of Features

Other domain models and the software architecture should be defined around the standard features. Alternative and optional features must be incorporated into the models and architecture, but should always be parameterized with the corresponding features so that instantiation of the models and architecture can be done easily.

5.1.4. Process and Guidelines

The feature analysis process consists of: (1) collecting source documents, (2) identifying features, (3) abstracting and classifying the identified features as a model, (4) defining the features, and (5) validating the model. The various sources of domain information are discussed in Section 3.2; each of the remaining activities is detailed below.

5.1.4.1. Feature Identification

Application features fall largely into four categories (as shown in Figure 3-1 on page 24):

- operating environments
- capabilities
- domain technology
- implementation techniques

Of these features, this method focuses on the analysis of features related to application capabilities. (This limitation is due to the experimental nature of the current project. The feature analysis process outlined in this section can be extended to cover other types of features.)

Features related to *capabilities* can further be categorized into three areas (according to [Myers 88]):

- functional features
- operational features
- presentation features

Functional features are basically "services" that are provided by the applications. Features of this type can be found in the user manual and the requirements documents. *Operational features* are those that are related to the operation of applications (again, from the user's perspective); that is, how user interactions with the applications occur. User manuals are a good source for this type of features. The last category, *presentation features*, includes those that are related to what and how information is presented to the users. User manuals and requirements documents usually provide information for this type of features.

The feature categories discussed above may not be complete; they are provided as a guideline for identifying features. Other types of features that are appropriate should also be included in the model. More experience in domain analysis supports the definition of a more complete set of feature categories.

The identified features should be named and any naming conflicts should be resolved. Synonyms for the features should also be recorded in the domain terminology dictionary.

5.1.4.2. Feature Abstraction, Classification, and Modelling

Once the features of all applications used in the analysis are named and defined and any naming conflicts are resolved, then a hierarchical model should be created by classifying and structuring features using the *consists-of* relationship. Whether or not each feature is mandatory, alternative, or optional should be indicated in the model. Each feature in the model should be defined. The description should also indicate whether it is a compile-time, an activation-time, or a runtime feature. This can be determined based on when and how frequently the adaptation will be made.

The classification of the features can be used in the components construction for modularization and for selection of appropriate development techniques. If the domain is well-defined and is expected to remain stable, a preprocessor or an application generator development technique might be appropriate to process the compile-time features. A table-driven approach which has been used in many terminal handlers (e.g., UNIX "termcap" files) might be appropriate for the activation-time features, and an interactive menu might be appropriate for the runtime features.

5.1.4.3. Model Validation

Whether the feature model correctly represents the features of the domain must be validated by domain experts and against existing applications. The domain experts who have been consulted during the analysis should *not* be selected for validation so as to avoid any possible bias. Also, at least one application that was not included in the analysis should be

used in the validation to determine the generality and applicability of the model. If available, a new *set* of several applications would provide a better validation, but the ability to do this will depend on the maturity of the domain and financial constraints.

5.2. Entity-Relationship Modelling

5.2.1. Purpose

The entity-relationship model captures and defines the domain knowledge that is essential for implementing applications in the domain. The domain knowledge is either contextual information which gets lost after the development, or is deeply embedded in software and is often difficult to trace. Those who maintain or reuse software have to re-acquire this knowledge in order to understand the problems that the application addresses. Therefore, the purpose of entity-relationship modelling is to represent the domain knowledge explicitly in terms of domain entities and their relationships, and to make them available for the derivation of objects and data definitions during the functional analysis and architecture modelling.

5.2.2. Model Description

The entity-relationship modelling technique in the FODA method is an adaptation of Chen's method [Chen 76] with semantic data modelling ([McLeod 78], [Borgida 84]). Chen's notation and method are used, with the adoption of the generalization and aggregation concepts from semantic data modelling that are used as predefined relationship types. Therefore, the basic building blocks of the model are entity classes and *consist-of* and *is-a* relationships. Entity classes represent abstractions of objects in the application domain and the relationship types *is-a* and *consists-of* represent generalization and aggregation relationships, respectively. Aggregation relationships specify composition structures between entities while generalization relationships specify commonalities and differences among entities. Any relationship types other than these two relationship types that are important for the domain may be defined using the form in Appendix A.4 and used in the model. The definitions of the *is-a* and the *consists-of* relationships are also included in Appendix A.4.

Relationships between entities are described graphically using [Chen 76]. An example of the entity-relationship diagram notation may be found in Figure 7-5 on page 62. Other information pertaining to each entity may be described textually following the sample form found in Appendix A.2.

More than one attribute may be defined for an entity. Also, attributes may be defined separately from entities and may be used in an entity definition. In this case, the values of an attribute used in the entity definition must be within the range of the attribute value specified in the original definition. Attributes used in the relationships must always be defined prior to use. An attribute should be defined using the form in Appendix A.3.

5.2.3. Model Usage

The primary use of an entity-relationship model is to support analysis and understanding of the domain problems and to derive and structure domain objects used in the applications development. Entities are units of domain information that have to be processed and/or maintained by the systems. They, with their definitions of attributes, can be used to identify domain objects, which are then used to define data-flows and data-stores in the functional model. They may also be used for object-oriented development.

The *consists-of* and *is-a* relationships generally are used to identify the compositions of domain objects and the commonality and differences among domain objects, which will lead to the development of domain products that are general and parameterized. For example, an inheritance structure may be derived from *is-a* relationships in an object-oriented development. Also, the *consists-of* relationships group together entities that are integrated. This information can be used to identify consistency and integrity rules and to derive data-stores definitions in the functional model. Often, integrated and related data are accessed together in applications and need to be kept in the same area.

5.2.4. Process and Guidelines

Construction of a system starts by perceiving entities and their relationships which may be derived from existing systems or from a hypothetical system. (An entity is either a physical entity or a concept.) Then, the perceived entities and their relationships are named, and each entity is characterized by its properties (attributes), some of which (for example, *name*) may be used as identifiers.

One of the heuristics used in the development of an entity relationship model is that entities, relationships, and attributes are usually English nouns, verbs, and adjectives, respectively. When analyzing the existing systems, nouns and verbs that repeatedly appear in the documentation are collected, synonyms are identified, and the meanings of the words in the context of the domain are clarified. The nouns and verbs that appear repeatedly in the documentation and are considered important conceptual elements for the domain are captured in the model as entities and relationships, respectively. This is a heuristic, however, not an algorithm, and should be treated as such. It is limited in its usefulness, and should not be applied indiscriminately.

The perceived reality may contain names which are not within the scope of the target domain: they are eliminated from the model. The concepts captured in the model constitute a major part of the domain terminology (which is recorded in the domain terminology dictionary).

The entities in the model are classified into homogeneous sets (classes or types) of entities; homogeneous in the sense that all entities in the same class have some properties in common. Each entity classified as such is named as an *entity type*, which is a unit in conceptual model construction. The generalization relationships identified during the classification will lead to *is-a* hierarchies. Other relationships existing between the entities, including the ag-

gregation shown by *consists-of* relationships, are also classified into relationship types, and these types are defined between the entity types. Properties of the entities in a class are classified into attribute types, and for each attribute type, possible values are defined. The entity-relationship model contains an abstraction of the target domain.

Of the many relationships that exist between entities, structural relationships are the most interesting, specifically *is-a* (generalization) and *consists-of* (aggregation). These relationships are important because they are critical to understanding similarities and differences. The entity-relationship analysis of the domain will primarily focus on these relationships; any other relationship types that are important for the domain may be added in the entity-relationship model, but only after they are defined using the form in Appendix A.4. The *is-a* and the *consists-of* relationship types are predefined and the definitions are also included in Appendix A.4.

5.3. Functional Analysis

5.3.1. Purpose

Functional analysis identifies *functional* commonalities and differences of the applications in a domain. It abstracts and structures the common functions in a model from which application specific functional models can be instantiated or derived with appropriate adaptation.

The feature model and entity-relationship model are used as guidelines in developing the functional model. The mandatory features and the entities are the basis for defining an abstract functional model. The alternative and optional features are embedded into the model during refinement. Also, any factors (other than features) that cause functional differences between the applications are defined as *issues* and *decisions*, which are used in the refinements for parameterization.

5.3.2. Model Description

Specifications of a functional model can be classified into two major categories: specifications of functions and specifications of behavior. The specification of functions describes the structural aspect of an application in terms of inputs, outputs, activities, internal data, logical structures of these, and data-flow relationships between them. The specification of behaviors describes how an application behaves in terms of events, inputs, states, conditions, and state transitions. (The sample domain analysis presented in Chapter 7 employs the Statemate *Activitycharts* and *Statecharts* to describe the functional and the behavioral aspects, respectively.)

An abstract model of the functionality of domain applications is defined at the top level. (Any difficulty in abstracting the functionality as a model might indicate that the selected domain is too broad.) During refinement of a model, there may be cases where an entity of a model can be refined in more than one way. For instance, in the case where there is more than one domain technology that can be selected, the selection of a particular technology can result in functional specifications that are different from others. The analyst must understand the implication of the selection of each technology and make a decision that is best for the

application. The major decision points and the alternative decisions at each decision point are captured as *issues* and *decisions* by the FODA method. (This concept is also supported by [Conklin 88], [Baldo 89], and [Bailin 89].) The *rationale* for each decision and any *constraints* or *requirements* derived from the decision are also recorded. The information that is collected for each *issue* and the related *decisions* is shown in a sample form found in Appendix A.

One thing that differentiates the FODA method from other domain analysis methods is *parameterization* through the use of *features* and *issues/decisions*. As an abstract model is refined, alternative and optional features are embedded into the model. Any issues raised during the analysis and the resolutions (i.e., decisions) of the issues are also incorporated into the model. There are generally three ways to incorporate the *features* and *issues/decisions* into the model:

1. By developing separate components (refinements) for every alternative, as illustrated by Case 1 of Figure 5-3.
2. By developing one component, but with parameterization for adaptation to each alternative, as illustrated by Case 2 of Figure 5-3.
3. By defining a general component and developing each alternative as an instantiation of the general component (with an inheritance mechanism) [Borgida 84], as illustrated by Case 3 of Figure 5-3.

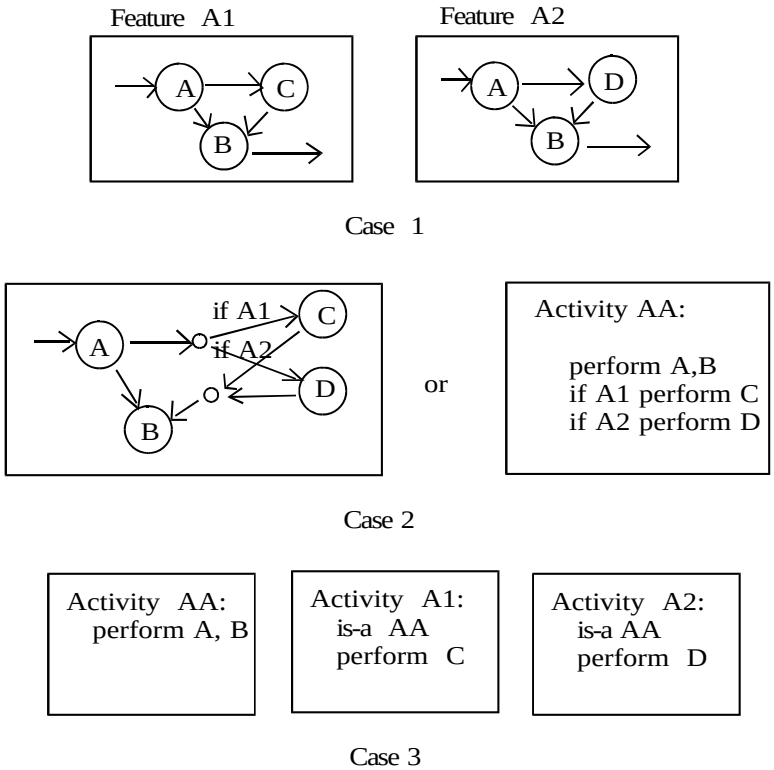


Figure 5-3: Parameterization: An Illustration

Most of the requirements analysis techniques available today do not support the above ap-

proaches, and it is especially difficult to extend them to support the second and third approaches. Statemate *conditions* are used in the example domain analysis to parameterize specifications as illustrated by Case 2.

5.3.3. Model Usage

The primary uses of this model are to (1) understand the domain problems and (2) reuse models in the requirements analyses. The model represents the functionality of applications from an abstract level down to the detailed level. The decomposition structure and rationales associated with decompositions will help analysts understand the domain problems. Also, analysts can reuse the model at the level where it is most appropriate for the given application.

5.3.4. Process and Guidelines

The notion of generalization/specialization is adopted to define generic functions and objects, and specifications of each system are made as specialization of the generic functions and objects. What to generalize/specialize can be determined as follows:

- Alternative features in the feature model may be used to identify generic functions. Alternative features are specializations of a more general feature, and the functionality corresponding to the general feature is defined as a generic function which is inherited by the functions implementing the alternative features.
- The generalization/specialization relationships (i.e., *is-a* relationships) of the entity-relationship model can be used to identify generic objects and the functionality associated with the generic objects.
- The context model identifies the external entities and the commonalities among the same type of entities, which can be used to define generalization/specialization of functions.
- Alternative domain technologies, with which different requirements decisions are made, can be a basis for defining generic functions. A generic function is defined based on the commonality of the alternatives, which is inherited by the functional definitions of specific technologies. This information is obtained by analyzing and comparing applications in the domain during the function analysis phase.

The process of functional model development takes both re-engineering and reverse engineering approaches to specify the functionality of existing applications. If requirements documents are available, the functional models are re-engineered from the documents. Otherwise, the functional models are reverse engineered from the design documents and/or code. In either case the functional models are specified using data flow and state transition modelling techniques (which is done in the Statemate notation for the feasibility study). The steps of the process are:

1. Gather and study the documents (e.g., requirements documents) describing the functionality of the applications. See if there is any standard model or if a model emerges as standard in the domain.

2. Produce data flow and finite state machine representations of the functional and behavioral aspects of each application used in the domain analysis. While naming objects in the representations, be certain that the semantics of the names are consistent; resolve any conflicts.
3. Based on the understanding of the models, see if there is enough commonality among the models to warrant a general model. If a general model cannot be readily abstracted because of structural differences, check if the entity-relationship model can serve as a basis for object-oriented modelling [Coad 89]. Check if the feature model or other real-world models such as the queuing network model, feedback control model, or decision support systems models can be used to represent the problem.
4. Refine the general model until all the problems represented in the application specific models are addressed. Embed features in the model to parameterize or to define alternative decompositions.
5. Check if all features are properly addressed in the model; document the mappings between features and the objects of the model.
6. Have the model validated by domain experts. Demonstrate the applicability of the model by using the model to describe an application not included in the analysis.

6. FODA Architecture Modelling

6.1. Purpose

The purpose of architecture modelling is to provide a software "solution" to the problems defined in the domain modelling phase. An architecture model (also known as a *design reference model*) is developed in this phase, and from it detailed design and component construction can be done.

A primary goal of the FODA method is to make domain products reusable. In the development of an architecture model, architecture layering is done so that reuse can occur at the layer appropriate for a given application and the impact of technical and requirements changes to the model can be localized.

A FODA architecture model is a high-level design of the applications in a domain. Therefore, the FODA method focuses on identifying concurrent processes and domain-oriented common modules, and on allocating the features, functions, and data objects defined in the domain model to the processes and modules. Many other implementation decisions still have to be made to complete the design.

6.2. Model Description

An architecture model must address the problems defined in the domain in a way that the model can be adapted to future changes in the problems and technology. This adaptation is achieved through architecture layering where:

- An architecture is defined at various levels of abstraction so that reuse can occur at the level appropriate for a given application.
- Packaging of domain functions and objects is done separately from packaging of implementation techniques so that:
    - Implementation decisions can be separated from the packaging of functionality.
    - The reusability of modules (both application-oriented and generic) can be increased.
    - The impact to the rest of the system arising from changes in implementation techniques can be localized.

Many decisions are made during the design of a software system. An application is decomposed into a collection of programs (i.e., processes) that can be compiled separately and executed in parallel. (A finite state machine model from the domain modelling phase provides the necessary information.) Each process must be designed as a hierarchy of modules with the allocation of functions and data objects defined in the data-flow model. Then, domain-oriented common modules that can be used across the applications must be identified to increase the reusability. Implementation decisions must be made in which various

implementation techniques such as communication and synchronization mechanisms, process scheduling methods, database management systems, and programming languages are selected.

A layered architecture of systems may be defined as shown in Figure 6-1. This is based on the types of design decisions discussed above and the general sequence of making those design decisions. The top, or *domain architecture layer,* is represented as a model showing the concurrent domain-processes and inter-connections between them. This model is called a *process interaction model* in the FODA method and is represented using the DARTS (Design Approach for Real-Time Systems) methodology [Gomaa 84]. The layer below that, the *domain utilities layer,* shows the packaging of functions and data objects into modules and the inter-connections between. This is called *module structure charts* and is represented using the Structure Chart notations [Yourdon 78] following the DARTS methodology. Domain-oriented modules that are common across the applications in the domain are also identified. For example, in the window management system domain (discussed in Chapter 7), window management library domain utilities (such as Xlib) are used for developing window applications, and these utilities are identified in this layer. The *common utilities layer* contains modules that can be used across different domains. Modules for inter-process communication and synchronization (e.g., a message queue implementation, an event handler) and for data management belong in this layer. Any utilities provided by the operating systems and programming languages, such as, semaphores and the Ada run-time environment, belong in the bottom *systems layer.* Classifications similar to Figure 6-1 may also be found in [Shlaer 90] and [Neighbors 87].

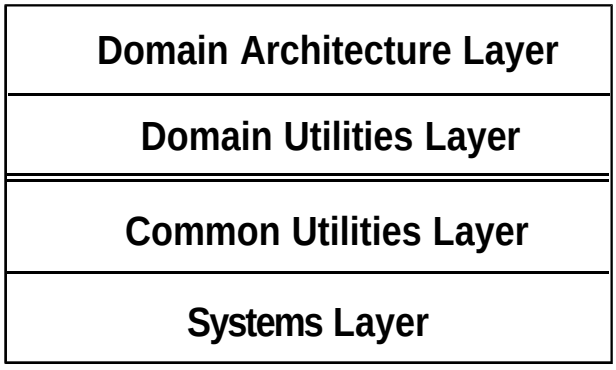


Figure 6-1: Architectural Layers

This methodology focuses on the top two layers, that is, the development of an application domain-oriented architecture. It is a high-level design where the packaging of functions and objects in software modules is the primary objective. Concurrent tasks are identified, and

communication and synchronization between the tasks are defined using the DARTS notation. Each task is designed as a sequential program by allocating application specific functions and data using the Structured Design. No decision as to the implementation of communication and synchronization mechanisms is made at this level; they should be made later to complete the design.

Components in each of the architectural layers above can further be layered based on the levels of "conceptual models" one can define for the component. For example, in the window management system domain, a layered design can be developed as shown in Figure 6-2. The layer at the lowest level, a virtual device driver, provides a conceptual model in terms of the pixel level operations hiding peculiarities of particular devices from the rest of a system. The next layer above the virtual device driver provides a conceptual model at the level of "graphics" where different types of lines and shapes are defined and the operations to create, move, and destroy those are provided. The layer above the graphics layer is defined based on the concept of windows and window operations: a window has a shape, contains information, has display attributes, etc.; operations for creating windows, displaying contents, and changing attributes are provided. The top level layer contains different types of windows (typically called *widgets*) including push buttons, scroll bars, menus, forms, etc., which can be composed to create more complex windows (i.e., *composite* windows).

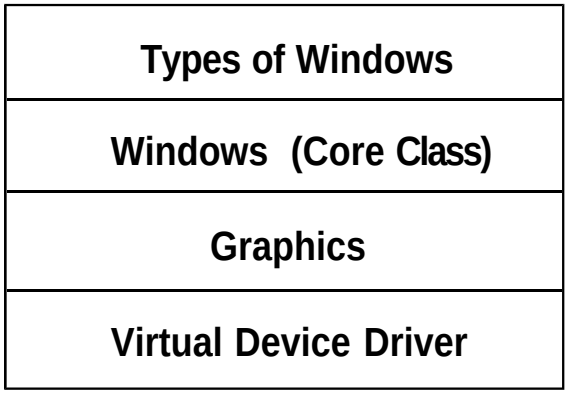


Figure 6-2: Window Management Subsystem Design Structure

The following should be noted from the above example.

- Each layer is defined based on a "conceptual model."
- A conceptual model at a low level is generic across more domains than the models above (i.e., each layer is a specialization of the layer below).

Designing a module based on a conceptual model is important because the model provides a basis for defining objects and operations and for verifying the design. Also, with an under-

standing of this model, users can easily relate this model to the problems they have and decide if the module implementing the model will solve their problems.

Although low-level models are generic and reusable across more domains than high-level models, the productivity increase from the reuse of high-level models is higher than that of low-level models (as explained in Section 3.1.3). Therefore, in the architecture modelling many levels of layering should be done to increase both productivity and reusability. However, program performance might be degraded with many layers and an optimal decision should be made considering all these factors.

As with other models discussed in the previous sections, features and other design decisions are embedded in the architecture model for parameterization. Also, many design decisions (e.g., selection of an implementation technique) are made during the design, which can result in different implementations. These decision points are captured as *issues* and possible alternatives are captured as *decisions* in the architecture model as discussed in Section 5.1.2. The types of information collected on design issues and decisions are the same as those collected on requirements issues and decisions (see Section 5.1.2 for details).

Packaging of functions and objects into modules must be done considering the processing time of the features (i.e., compile-time, activation-time, and runtime) that each module implements. Packaging of compile-time features must be done so that modules implementing each compile-time feature can be identified uniquely from a collection of alternative modules or be instantiated from general ones. There must be a module for activation-time features that collects values of the activation-time features at the start of execution, verifies the correctness, and stores values for other modules to access during the execution of the application. For runtime features, a module(s) allowing interactive selection of the features during execution must be included in the design.

6.3. Model Usage

An architecture model can be used to:

- Make a detailed design and identify opportunities for reusable components.
- Provide a reference model for future systems development and for evolution of existing systems.
- Ascertain reusability of candidate components.
- Provide a model for managing (classifying, storing, and retrieving) software components in a domain.
- Provide a framework for tooling and systems synthesis.

An architecture model for a domain serves many different purposes. It is in essence a standard reference model for building applications in the domain, and thus may be used as a framework for building new systems, as a means to educate potential system engineers in that application domain, or as a template for the construction of domain-specific reusable components.

6.4. Process and Guidelines

FODA uses the DARTS methodology to develop an architecture model with some extensions for parameterization of modules as illustrated in Figure 5-3. (The DARTS methodology uses the Structured Design [Yourdon 78] technique to design module structures. Therefore, the Structured Design technique will be considered a part of the DARTS methodology.) The DARTS methodology is summarized here; details can be found in [Gomaa 84].

- Real-time software consists of a set of synchronously or asynchronously communicating tasks. Each task is a sequential program.
- From data flow diagrams, concurrent tasks are identified using the following criteria:
    - dependency on I/O devices
    - time-criticality
    - computational requirements
    - functional cohesion
    - temporal cohesion
    - periodicity
- A Task Communication Module (TCM) handles all cases of communication among tasks. Two types of TCMs are supported in DARTS: message communication modules for loosely or tightly coupled message communication, and information hiding modules for data pool or data store.
- A Task Synchronization Module (TSM) is typically the main module of a task, controlling the synchronous behavior of the task. A task may wait for one or more synchronizing or message queue events.
- Each task is designed using the Structured Design technique.

A process for developing an architecture model is defined below based on the DARTS methodology, the architecture layering concept, and the parameterization concept. The process is:

1. Identify major concurrent processes from the data-flow diagrams and finite state machine diagrams. A Statemate representation of the finite state machine identifies parallel states which indicate concurrency.
2. Identify other concurrent processes using the criteria provided by DARTS.
3. Allocate the functions and data to each process and check if all the functions and data in the data flow diagrams are properly allocated to the processes.
4. Define process interactions using the DARTS notation.
5. Design each process using the Structured Design method producing structure charts. (Other design methods may be used instead of the Structured Design method.) Each process consists of a TSM (as the main module), which invokes TCMs and application modules. Application modules include abstract data types, mathematics libraries, device interface modules, and other information hiding modules. In designing application modules apply the layering

concept as shown in Figure 6-2. Also, define modules for activation-time and runtime features.

6. Parameterize modules for the compile-time features (as shown in Figure 5-3) so that they may be instantiated for different selections of compile-time features.
7. Specify the components identified in the structure charts. Specifications of each module should include operations and parameters, a high-level internal logic, and allocation of features and functions. Ada PDL [IEEE 89] may be used for this purpose.

7. Application of Domain Analysis to Window Management Systems

The previous four chapters have presented the underlying concepts and specific products of the FODA method. This chapter presents a comprehensive example of their application.

As a sample domain *window manager software*, a sub-domain of *window management systems software,* offers a fairly complex set of requirements for domain analysis and is an application area that should be familiar to most users of the method. This domain features:

- Availability of many examples of implementations.
- Relevant literature and documentation, both user and developer.
- Availability of domain expertise to those performing the example analysis.

This chapter presents an application of the FODA method to the *window manager software* domain. An overview of window manager capabilities is presented, followed by presentations of each of the domain analysis products produced in the course of the study. Due to the size of even this sample domain, only excerpts of some of the models are discussed. For the complete documentation of the analysis, see Appendices B-G.

The example covers the first two phases of the FODA method: context analysis and domain modelling. The context analysis section focuses on the window manager context diagram and structure diagram. (However, not all of the steps outlined in Section 4.4 were performed due to resource constraints.) The domain modelling section describes the window manager entity-relationship diagram, the different components of the feature model (feature diagram, composition rules, and "issues and decisions") and the automated features tool, and the functional behavior model's data flow and state transition models. Although the architecture modelling phase of the method was not applied to this sample domain because that portion of the methodology had not been completed, the results of the first two phases demonstrate a successful application of much of the method.

7.1. Definition of a Window Management System

A *window management system* (often called simply a *window system*) is a type of interactive user interface that enables users to work with multiple separate applications at the same time. This is achieved through the use of a *desktop* metaphor, in which each process is associated with a graphical *window,* which is visually analogous to a paper or document on a physical desktop. Users may switch back and forth between different applications much as a person at a desk might choose between different documents on a desktop in the course of a day. A window management system provides the functionality to create and manipulate this display of multiple processes.

The functionality of a *window manager* is central to a window management system, and is to help the user manage screen "real estate," i.e., the portions of the screen that are available

for displaying useful information. The window manager helps the user to do this by providing various operations on windows that are, again, taken from the desktop metaphor.

For example, the window manager allows users to make (*create*) and discard (*destroy*) windows. Users can relocate (*move*) windows to different positions on the screen just as they might move documents on a desktop. Some window managers allow users to stack (*expose* and *hide*) windows on top of one another. Most window systems extend the desktop metaphor by allowing users to change the size of windows (*resize*) and to change (*iconify*) the windows into *icons*, an alternate form of the window that requires little space. With these operations available the user is in full control of the appearance of the workstation screen and may effectively manage several different tasks. Figure 7-1 shows a conceptual view of where the window manager is positioned relative to the "outside world." It accepts user inputs from the pointer (mouse) and the keyboard, communicates with the appropriate independent client programs, and sends the output to the display [Peterson 86].

Despite the fact that the window manager in some sense has control of the user interface, it does not directly influence the applications that run in the windows; it simply routes the input and output to and from the application. Often a window manager has no knowledge of the contents or activities of an application within a window. As an example of this, applications are often responsible for maintaining the contents of their own windows. The window manager will notify an application that its display has been damaged (i.e., by having a previously obscured portion revealed), but the application is responsible for redrawing it. Even if the window manager *does* preserve obscured portions of windows for the applications, it is simply saving graphic data.

There are many different implementations of window managers and some have important differences, but there is still significant commonality. Some window managers do not allow windows to be stacked on top of one another (*tiled* systems), while most others do (*overlapped* systems).[9] Other window managers do not support icons, or have more subtle differences in the way the common operations work. Nonetheless, virtually all window managers support creating, destroying, moving, resizing, and other window operations.

An important aspect of window management systems is the distinction between the *capabilities* of the window system and the *presentation style*, or set of conventions, by which the window system operates. Often applications are written under a window system where a consistent appearance, style, and interface, referred to as a "look and feel", are considered more important than the functionality of a single application. The window manager is an important part of the "look and feel" of a window system, primarily supplying the "feel." Of the window managers that are currently available, many have been written to run under X windows, since X allows a "user-defined" window manager that is separate from the rest of the window system. The following list of window managers displays a wide range of functionality, all implemented with the X window system.

[9] In fact, a window manager may support virtually any window layout policy the designer wishes to impose.

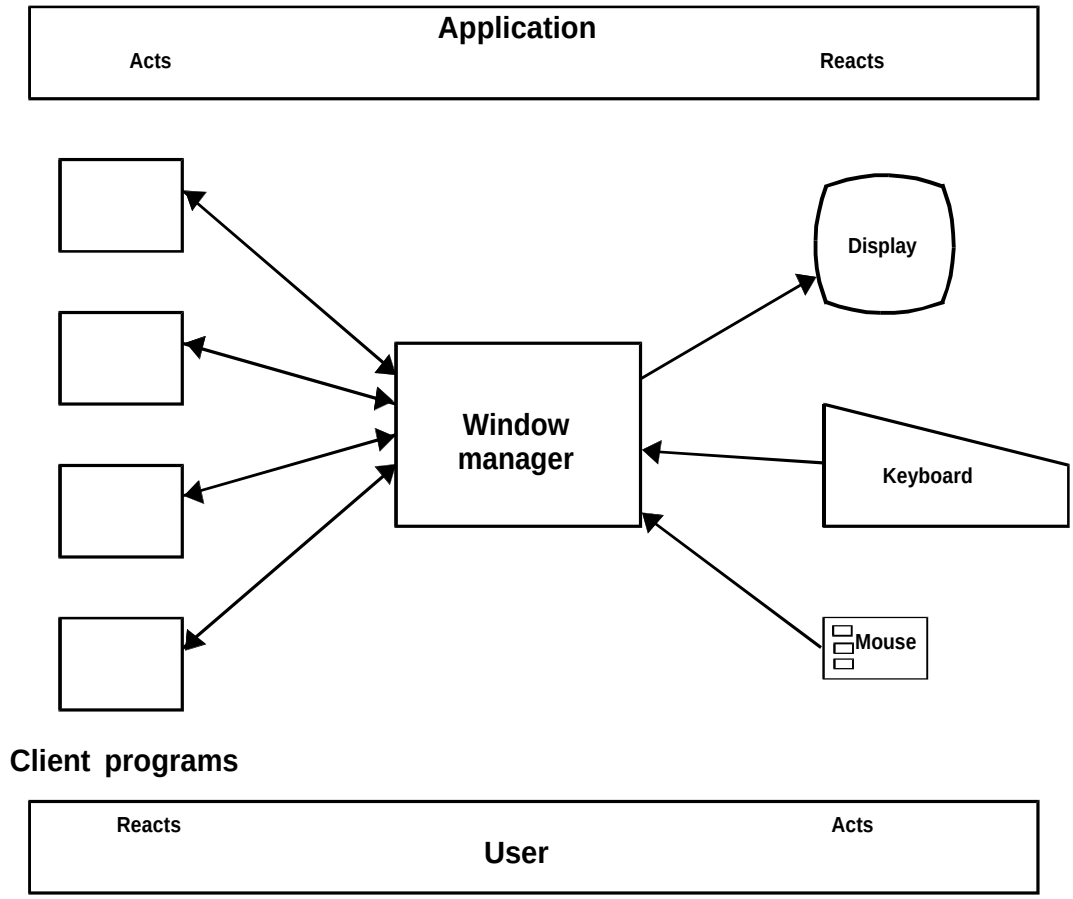


Figure 7-1: Function of a Window Manager

- uwm: *Universal Window Manager*
- olwm: *Open Look Window Manager*
- wm: *Andrew Window Manager*
- mwm: *Motif Window Manager*
- wmc: *CMU Computer Club Window Manager*
- twm: *Tom's Window Manager*
- gwm: *Generic Window Manager*

In other systems the window manager is integrated into the window management system and has no individual name, as is the case in the Macintosh, SunView, the Symbolics window system, and others.

7.1.1. Window Manager Capabilities

In order to be able to discuss the capabilities of window managers it is necessary to review some of the basic functionality. Some of the operations which are typically, but not necessarily, provided by a window manager are:

- *Create* and *Destroy*
- *Move* and *Resize*
- *Iconify* and *Deiconify*
- *Hide, Expose,* and *Circulate*
- *Change Focus*
- *Refresh*

For the purposes of explaining the FODA method in the context of the window manager, the examples will be centered around the window manager *Move* operation for windows. *Move* is a *mandatory* operation, and must be present in all window managers. This restriction in scope of the presentation of the analysis is done to limit the amount of new concepts and terminology which must be introduced. *Move* was chosen because it is one of the most complex operations a window manager must perform; as a result it has a wide variety of feature variations. The full set of features and other results for all window manager operations may be found in Appendices C-G.

Figure 7-2 depicts several common window manager features that pertain to the *Move* operation. These are defined below.

`constrainedMove:`	A window may be constrained so as to move only horizontally or vertically, rather than in any direction.
`eraseBefore/eraseAfter:`	The image of a window is erased either *before* the move operation begins, or *after* it ends.
`exposeAfterMove:`	After the *move* operation ends, the window is placed at the top of the *stacking order*[10] (in an `overlappedLayout` system).
`ghostFeedback/opaqueFeedback:`	Either a "rubber-banded" outline of a window (ghost), or the entire window image (opaque) is displayed while the position of the window is being changed.
`moveIcon:`	This feature provides the ability to move an icon to a new location.
`objectAction/actionObject:`	(also called *select-then-operate* or *subject-verb*) This feature determines if the user must first select an object to operate on (a window) and *then*

[10]The order of overlapping sibling windows that determines which one lies visually on top and which on the bottom. *Circulating, exposing,* and *hiding* windows changes the stacking order. The first window in the *stacking order* is the window on top.

	an action to perform on it (`objectAction`), or vice versa.
`overlappedLayout/tiledLayout:`	Either windows may overlap (or be overlapped by) other windows, or no windows are allowed to overlap (like tiles).
`partiallyOffScreen:`	A window is allowed to be pushed partially (but not entirely) off the screen.
`realEstateMode/listenerMode:`	(also called *point-to-type* or *follow-the-pointer*) The `realEstate` feature is a paradigm where the *keyboard input focus* is always at the window in which the pointer is currently "in." In `listenerMode` (also called *click-to-type*) the input focus is set to be a particular window by a mouse click, regardless of the position of the pointer.
`zapEffect:`	Optional "ghost" lines which temporarily "flash" to follow a window or icon from its original position to its new position after a *move* operation.

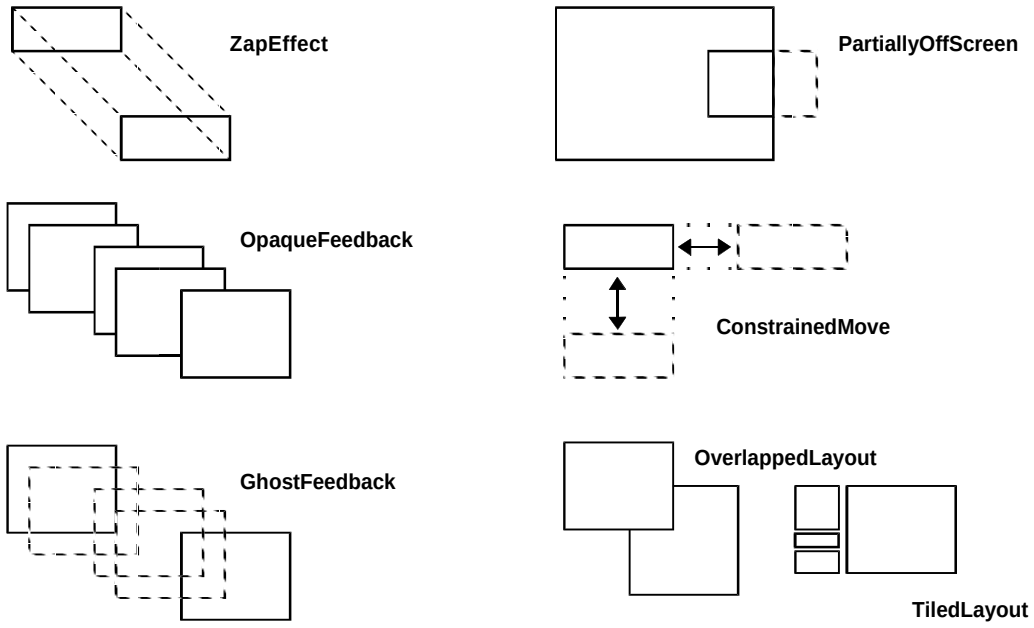


Figure 7-2: Sample User Features Found in Window Managers

7.2. Scoping the Window Management System Domain

The first step in performing any domain analysis is to gather sufficient information about the domain. The following sources were used as inputs to the window manager domain analysis:

Domain experts: people with extensive knowledge of user interfaces and window management systems, and experienced users of these systems.

Window systems: user experience primarily with features of X10/uwm, VMS Windows, and SunView, with additional experience with X11/uwm, Macintosh, Andrew, Symbolics, OSF/Motif, and NeWS

Domain literature: books, articles, surveys, manuals, and evaluations of many different window managers and window management systems

It is essential to first thoroughly understand the proposed domain area in order to properly scope the domain. A lack of sufficient domain knowledge can lead to choosing domains that are too large, have relatively little commonality, or do not have clean, logical boundaries to their scope.

After an initial review of the window management system domain, it became clear that an analysis of the entire domain was inappropriate for a small-scale feasibility study. It also became clear that the *window manager* portion was central to the user-visible functionality of the window management system. Further study of window managers confirmed this, and the window manager itself became the scope of the feasibility study domain analysis.

Given a well-defined domain, at this stage of the process it is most useful to have access to any previously done surveys or commonality studies of applications in the domain. Such surveys do exist for many domains. In the window management domain, a paper by Brad Myers, "A Taxonomy of Window Manager User Interfaces" [Myers 88], provided an excellent starting point for further research, as well as an initial bibliography. Given a basic background in the domain, other references could be researched for additional information before having discussions with domain experts.

A significant advantage in collecting information about external, user-visible aspects of window managers came from the ready availability of multiple implementations of window managers. While this availability of systems is not the case in some other domains, there are always "existing systems" of some kind, even if they are manual systems and/or procedures.

In terms of the FODA method, scoping of the domain to be analyzed is done in the *context analysis* phase. The two primary products of this phase are the *structure diagram* and the *context diagram*. The purpose of these diagrams is to clearly delineate what the scope of the analysis is to be. It is useful for the domain analysts to return to these diagrams throughout the analysis as they help to avoid unnecessary and wasteful digressions.

The structure diagram (shown in Figure 7-3) shows the position of the window manager within the entire domain of window management systems. The window manager is shown

in bold outline in the upper left corner. The purpose of this diagram, informal though it may be, is to separate the logical concept of the window manager from many other related items with which it is often confused. For example, a window manager is *not* a graphical user interface (GUI), although it may provide the "feel" portion of a GUI's "look and feel".

The context diagram (shown in Figure 7-4) is a standard top-level data flow diagram of the interfaces a window manager has with the other significant parts of a window management system. This particular view of the role of a window manager within a window system was one of the products of the early context analysis process, and was not taken from any outside source. In the context diagram the functionality of the window manager is separated from the closely related functions of the input manager, the process manager, and the display manager. The input manager converts the "raw" user input events into higher-level events for the window manager.

The implications of this arrangement are that the window manager is not responsible for interpreting combinations of keys and button presses, dealing with application processes, or maintaining the integrity of the screen. The window manager *is* responsible for allowing the user to manipulate the shape, size, and position of windows. These other (admittedly closely related) activities are *not* part of the window manager functionality and are therefore not part of the window manager domain analysis. While the context diagram serves as a good cut at the boundaries of the domain analysis scope, the process of refining the scope continues throughout the analysis.

7.3. Domain Model

The *domain model* (as explained in Chapter 5) describes the elements of systems in a given application domain from the point of view of a "problem space"; that is, *what* the systems in that domain must do. It complements the *architecture model* (i.e., the *solution space*) which describes various alternative ways in which systems may be built to meet the requirements of the domain model and the operating environment.

The domain model comprises the following:

- entity-relationship model
- feature model
- functional model
- domain terminology dictionary

The domain terminology dictionary is used to standardize the terminology that describes the domain. The dictionary is especially critical in a new and rapidly evolving domain, such as window management systems, where no clear standard may exist. As an example, even the term "window" is non-standard, also being referred to as "view," "canvas," and "wob" in different window system documents. The dictionary results from information gathering in all of the analysis phases. The dictionary resulting from the window manager domain analysis has approximately 300 entries including many synonyms due to the evolving window system nomenclature. This dictionary is included in this report as Appendix B.

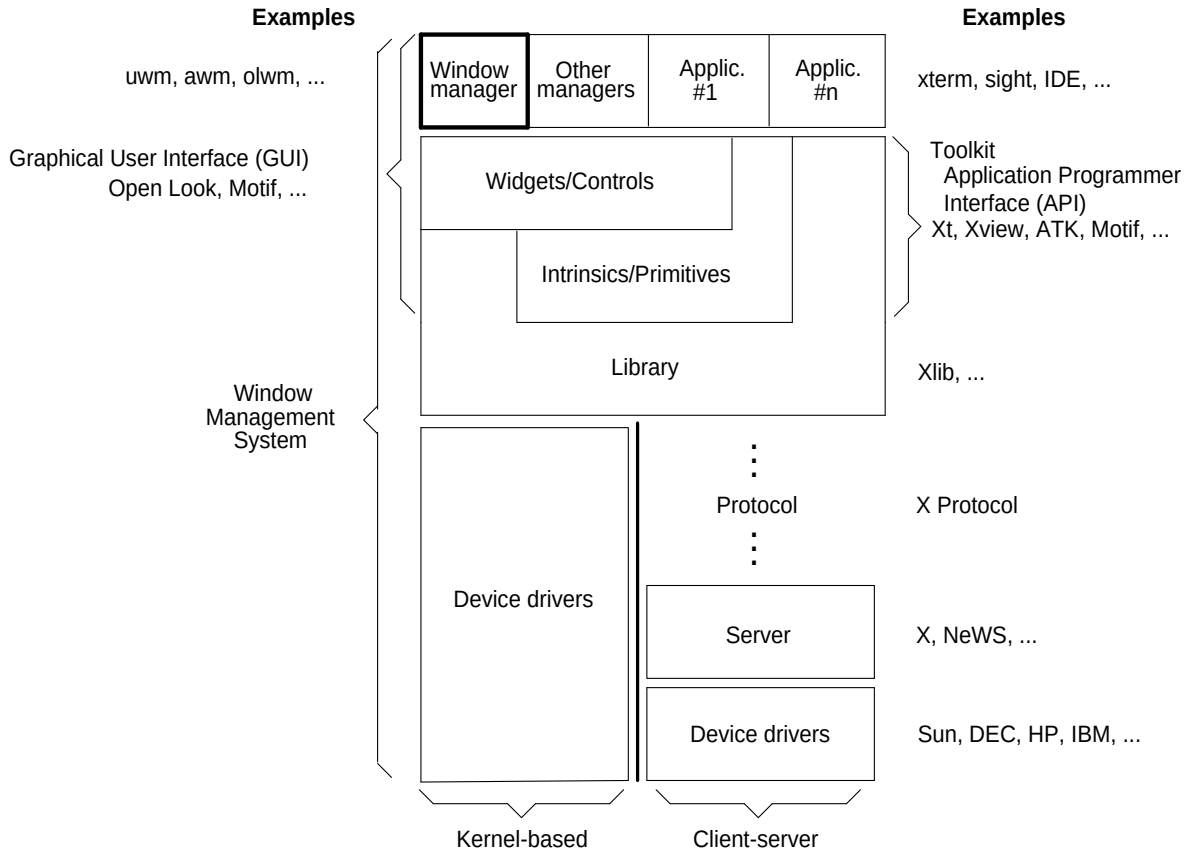


Figure 7-3: Structure Diagram: Relationship of Window Managers to Window Management Systems

The following three sections describe the three components of the domain model for the window manager domain analysis: the entity-relationship model, the feature model, and the functional model.

7.3.1. Entity-Relationship Model

The entity-relationship model used in the domain analysis consists of three parts:

1. entity-relationship diagram
2. attributes of the entities
3. constraints on the entities and relationships

Figure 7-5 is the entity-relationship diagram for a window manager, showing (in a different way from the context diagram) its relationships with the other significant entities of window systems in general. These entities are (as shown in Figure 7-5):

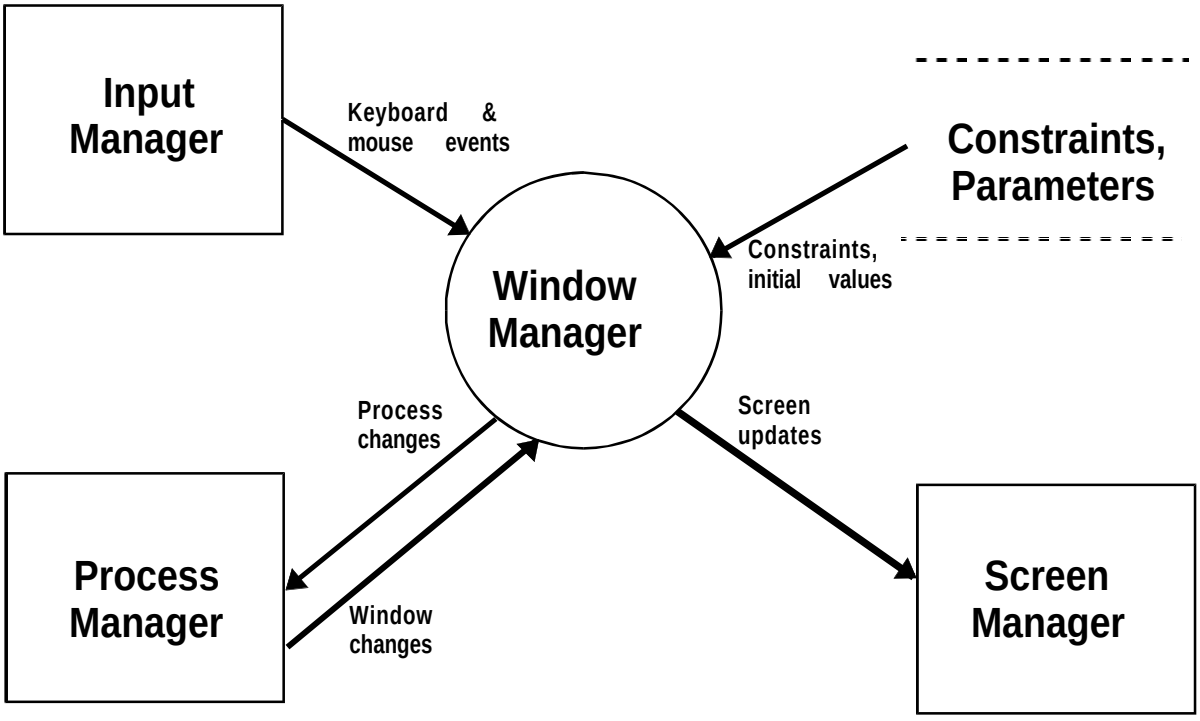


Figure 7-4: Context Diagram: Major Data Flows of Window Management Systems

- window
- main window
- process
- icon
- screen
- pointer

Many of the relationships shown in the diagram are straightforward, such as those between the pointer,[11] the screen, a window, and the application process(es). Some of the other relationships require further explanation. By its nature this diagram is a generalization of the way many window systems are structured, and therefore the degree to which it maps directly to the model of a specific implementation varies. The diagram uses the *is-a* and *consists-of* relationships defined in Section 5.1.2 to show that both an icon *is-a* window, and a "main window"[12] *is-a* window. The diagram calls out these two entities as specializations

[11]The 1-to-many relationship between the screen and the pointer is a maximum. While this is almost always a 1-to-1 relationship in practice, the emerging window manager technology indicates that this may change.

[12]A *main window* is a window displayed in its normal, full form, i.e., *not* an icon.

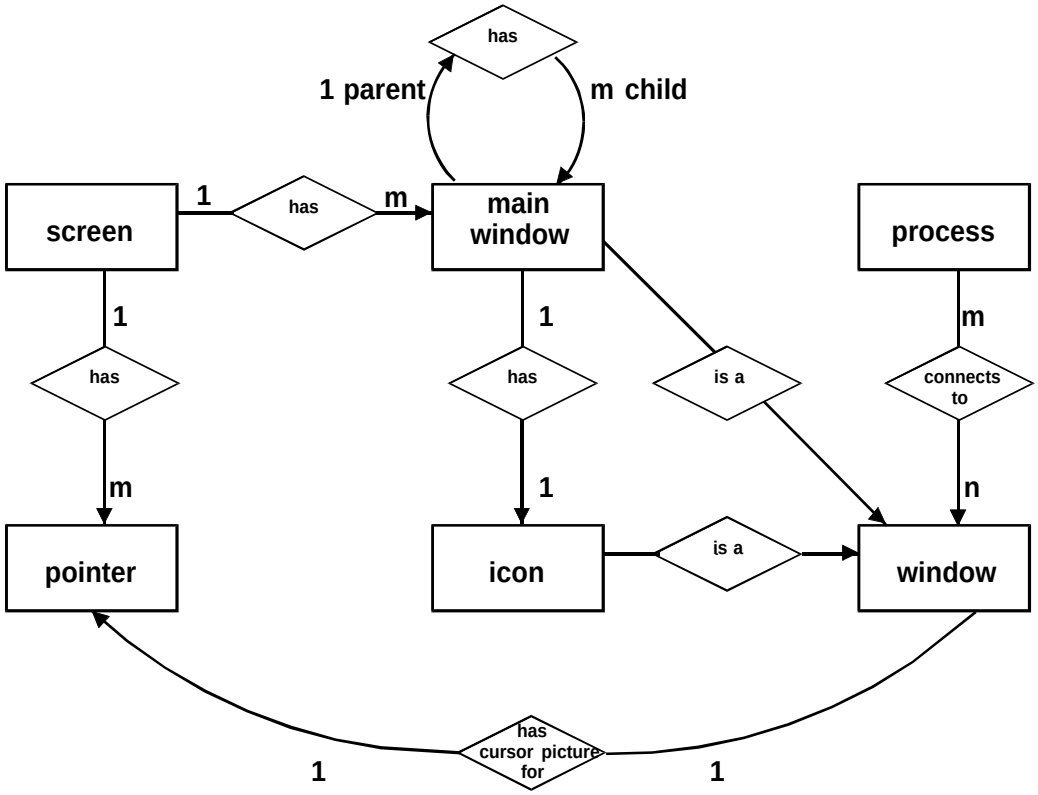


Figure 7-5: Window Manager Entity-Relationship Diagram

of a generic window entity because they are closely related, and they are fundamental to window managers. While some implementations explicitly view icons as a type of window (such as X Windows), others do not. However, the fact that a particular implementation does not treat icons as a type of window does not invalidate the application of the entity-relationship diagram to that implementation.

Another relationship depicted in the diagram is the one-to-many, parent-child, *has* relationship among main windows, which describes a parent-child hierarchy. While this is not always the approach used in actual implementation, it is useful to describe the behavior when one window manager operation can affect a group of seemingly independent windows.

The attributes associated with these entities are listed in Appendix A. The attributes could be used in designing the data structures of the implementations of these entities, much as they are traditionally used in database schema design. One assumption that is made in the listing of these attributes is that inheritance of attributes will take place across *is-a* relationships. As a result, attributes belonging to the generic window entity are not also listed as belonging to icons or main windows.

In addition to the entity-relationship diagram and the attributes, during the course of the domain analysis it became clear that there are many (typically unstated) constraints on the operation of a window manager. The entity-relationship *model* can include these constraints

as well, even though they are not expressible through the entity-relationship diagram. These constraints include:

- There is at most one input focus.
- There is one *root* window that has no parent window.
- The *root* window exactly covers the screen.
- The pointer cannot leave the area of the screen.
- Events propagate from the *source* window to *ancestor* windows until they are handled.
- Child windows exist only while their parent windows do.

Some constraints are specific to the presence of certain features in the window manager. In such cases the entity-relationship model may be parameterized by the feature model, as are many other models of the domain model.

If the `windowLayout` is `overlappedLayout` then:

- A child window must be in front of its parent window.
- Only *sibling* windows may overlap.
- Child windows are clipped by their parent window.
- The position and size of child windows within their parent window can be fixed.

7.3.2. Feature Model

The purpose of the feature model is to describe the "requirements space" of known window managers. The model should encompass as many window managers as is feasible, to include the fullest range of features and feature values. A specific implementation of a window manager, such as the OSF/Motif mwm, may be thought of as an *instantiation* of the feature model, or a set of *feature values* which describes its particular capabilities. Being able to describe a proposed system to a potential customer in terms of the possible features which can be provided simplifies the requirements elicitation process, and can clarify the various implicit trade-off decisions which must be made.

The components of the feature model are as follows:

Feature diagram:	A graphical And/Or hierarchy of features
Composition rules:	Mutual dependency (Requires) and mutual exclusion (Mutex-with) relationships
Issues and decisions:	Record of trade-offs, rationales, and justifications
System feature catalogue:	Record of existing system features

The following paragraphs discuss each of these parts of the feature model in detail.

7.3.2.1. Feature Diagram

The feature diagram, shown in Figure 7-6, is an *and/or* tree of different features. *Optional* features are designated graphically by a small circle immediately above the feature name, as in `partiallyOffScreen`. *Alternative* features are shown as being children of the same parent feature, with an arc drawn through all of the options, as is the case in `windowLayout`. The arc signifies that one and only one of those features must be chosen. The remaining features with no special notation are all *mandatory*.

The line drawn between a child feature and a parent feature indicates that a child feature *requires* its parent feature to be present; if the parent is not marked as valid, then the child feature for that system is in essence "unreachable." For example, if the `windowLayout` were selected to be `overlappedLayout`, then the feature `tiledColumns` would be "unreachable" for that specific system, since its parent `tiledLayout` would not be valid.

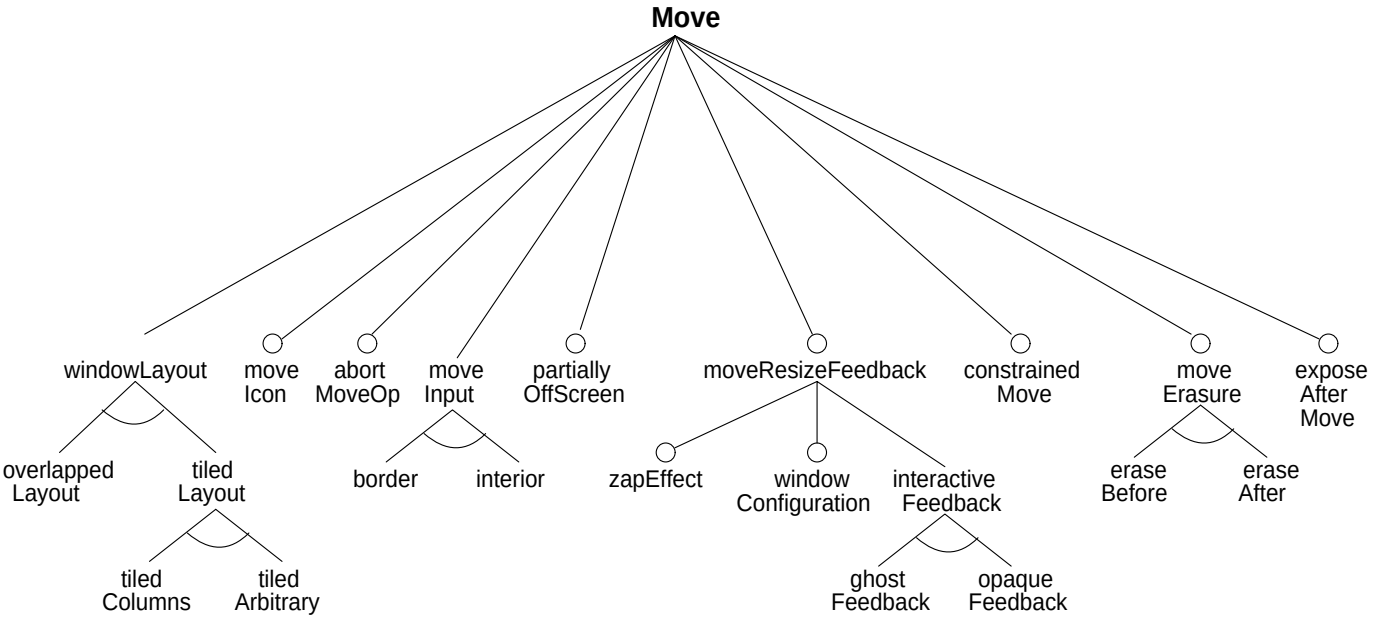


Figure 7-6: Features for the Window Manager Move Operation

To illustrate the use of the feature diagram Figure 7-7 shows a comparison of the *move* operation features for two different existing window managers: X10/uwm and SunView. The selected optional and alternative features are highlighted in the diagram with boxes. For example, notice that the feature `partiallyOffScreenWindows` (abbreviated on the diagram) is present in X10/uwm, but not present in SunView. Thus, when a SunView window is moved so that its border touches the edge of the screen, the window will stop moving in that direction. In X10/uwm the window will continue to move, disappearing off the screen, until the cursor hits the screen edge and stops the window from moving completely off.

This type of comparison information, which may be available in this graphical form or in the catalogue form shown in Appendix C, makes the task of evaluating and comparing different

systems straightforward. Certain types of information are more difficult to obtain from such a display, such as knowledge of invalid feature combinations or underlying issues and rationales. These types of information are discussed in the next two sections.

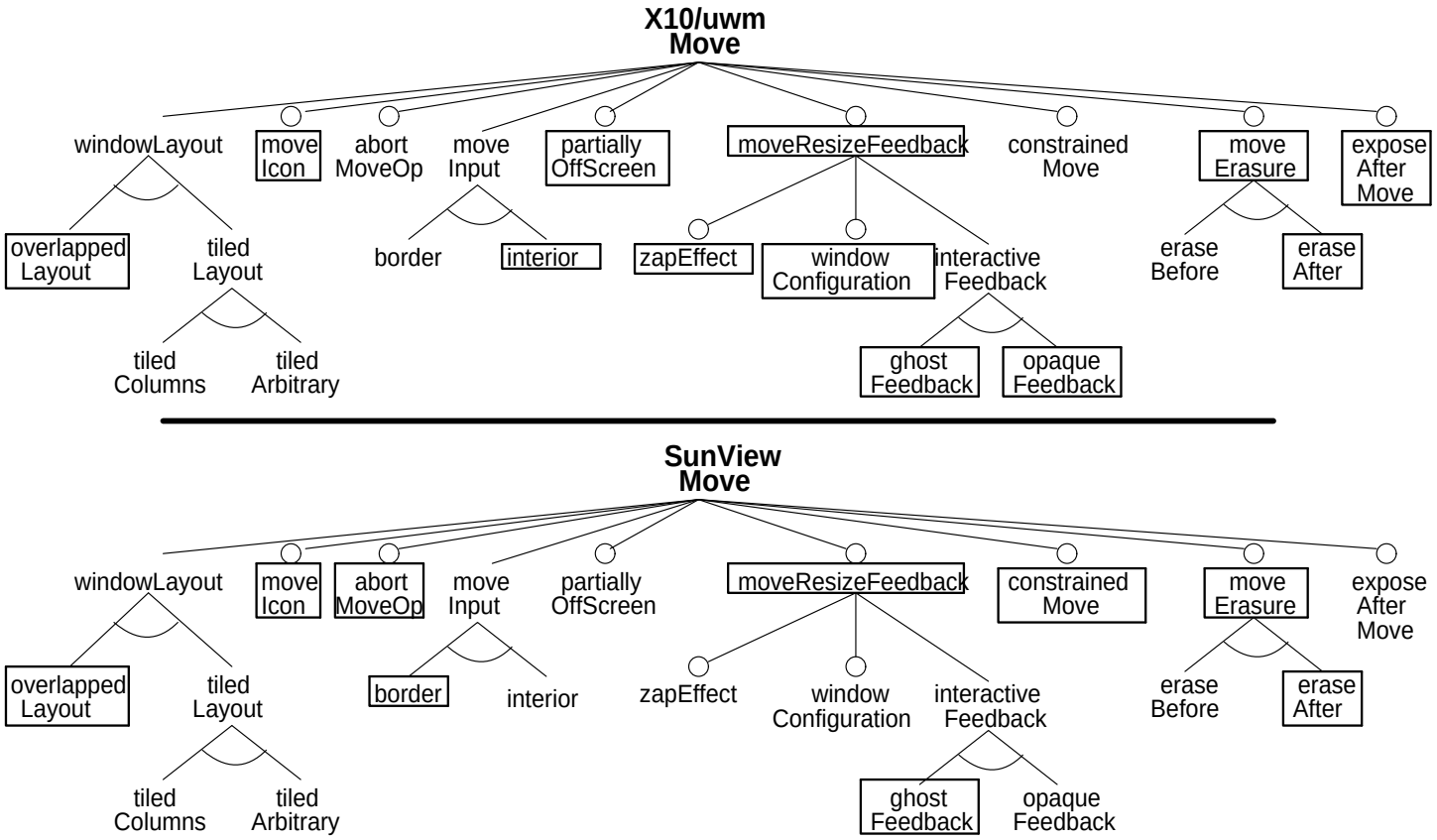


Figure 7-7: Comparison of Move Operation Features in X10/uwm and SunView

7.3.2.2. Composition Rules

Features are related to one another primarily through the use of *composition rules,* which are a type of constraint on the use of a feature. Composition rules have two forms: (1) one feature *requires* the existence of another feature (because they are interdependent), and (2) one feature is *mutually exclusive with* another (they cannot coexist).

The textual representation for these rules is as follows:

```
<feature1> (‘requires’ | ‘mutex-with’) <feature2>
```

An example of a composition rule used in the window manager domain is:

`moveIcon` requires `hasIcons`

In Section 7.1.1 these window manager capabilities were defined. Composition rules may be obvious, given an understanding of the domain. In this case a window manager cannot have

the moveIcon feature unless the system hasIcons. While this is clear, in large system definitions these interdependencies can be lost in the overall complexity. Also, some composition rules are less immediately obvious, but are based on common sense. For example:

opaqueFeedback mutex-with moveErasure:
: If the entire window image is moved, then there is nothing left in the previous position to erase.

zapEffect requ ires ghostFeedback:
: If the entire window image is moved (i.e., opaqueFeedback), then there is no window to draw the final zap lines from.

zapEffect requires eraseAfter:
: If the old window image were erased before the *move* operation, then there would be nowhere to draw the final zap lines from.

ghostFeedback requires moveErasure:
: If ghostFeedback is used, then the old window image must be erased at some point, either before or after. Thus one of the two alternatives of moveErasure must be selected.

exposeAfterMove requires overlappedLayout:
: An *expose* operation can only be done in an *overlapped* system.

These composition rules relating the features were often derived from experience with systems that have these features. For example, observing the interaction of ghostFeedback and moveErasure makes the composition relationship clear.

7.3.2.3. Issues and Decisions

A record of the issues and decisions that arise in the course of the feature analysis must be incorporated into the feature model to provide the rationale for choosing options and selecting among several alternatives. As an example, the interactiveFeedback feature has two different alternatives: ghostFeedback and opaqueFeedback. It is impossible to select one or the other without having access to the same information the original designer had. However, if the feature model contains a record of the original rationales, it is a simple process. The following excerpt from the "Issues and Decisions" record attached to the interactiveFeedback feature[13] demonstrates the usefulness of this information.

[13]The forms for recording this information (presented in Appendix A) were modified for use with the window manager example.

Issue: Resource consumption/Feedback clarity

Description: `interactiveFeedback` is the way the window manager shows the user the current size or shape of a window being moved or resized.

Raised at: `interactiveFeedback`

Decision: `ghostFeedback`

Description:	An outline and/or grid of the window is drawn and moved with the cursor to the new location, where the complete window is drawn (and the old window erased).
Rationale:	Provides sufficient user feedback for positioning and resizing, and requires significantly fewer resources than redrawing the entire image often enough to follow the moving cursor.

Decision: `opaqueFeedback`

Description:	The window manager moves or resizes the entire original image of the window.
Rationale:	Opaque moving and resizing allows the user to see immediately what the window will look like in the new position or shape, and is typically used on fast displays where the act of updating a potentially complex window display is feasible.

If the user knows enough about the trade-offs between the two options, then an informed decision based on the user's hardware environment (in this example) may be made.

7.3.2.4. System Feature Catalogue

In the course of gathering information for the domain analysis, one useful source is experience with existing systems in the domain. This is not always an option, as some domains may have a history of largely manual methods, but in the case of window managers there are many existing systems. It is important to record the features and feature values of actual existing systems (even in the case of manual methods) to allow for later modelling of the systems in terms of their features. An excerpt of the system feature catalogue generated for the window manager domain analysis is given in Table 7-1 for the *Move* operation features.

System/Feature	X10/uwm	VMS Windows	SunView	Mac Windows
moveWindow	*Move*	yes	*Move*	yes
constrainMove	no	no	yes	no
moveIcon	yes	yes	yes	no
moveErasure	after	before	after	
exposeAfterMove	yes	yes	no	yes
zapEffectMove	*	no	no	no
interactiveFeedback	*	ghost	ghost	ghost
partiallyOff	yes	yes	no	yes
abortMove	no	no	*Cancel*	no
selectOrder	actionObject	objectAction	objectAction	objectAction

Table 7-1: Window Operation Functionality

Certain window managers have already incorporated into their software some of the range of variation in the domain, making it possible for the user to specify certain features of the appearance and functionality at activation-time or runtime. This is specified in Table ref{Catalogue} by an asterisk ("*"). The X11/twm window manager profile file offers a number of user-settable options, many of which relate directly back to features offered in the feature model. Some of these features are illustrated in Table 7-2.

In the full feature catalogue tables in Appendix C, as well as in Table 7-1, the following conventions are used:

- <Name>: This is the name used for this feature on this system (i.e., *Move* or *Cancel*).
- "yes/no": This feature does/does not exist on this system.
- <blank>: No information has been collected for this feature on this system.
- "*": This feature is bound at either activation-time or runtime, but not at compile-time.
- "--": This feature is inapplicable to this system (a composition rule with another feature excludes it).

Profile Option	Feature
AutoRaise {"*application*"}	`exposeAfterMove`
DontMoveOff	no `partiallyOffScreen`
DontIconifyByUnmapping	`unmappedDeiconifiedIcons`
IconManagers {"*application*"}	`iconBox`
NoHighlight	no `highlightInputFocus`
NoRaiseOnDeiconify	no `exposeAfterDeiconify`
NoRaiseOnResize	no `exposeAfterResize`
NoTitle	no `titleBars`
NoTitleHighlight	no `titlebarHighlight`
OpaqueMove	`opaqueFeedback`
StartIconified	`createIconified`
WarpCursor	`warpToWindow`
Zoom 10	`zoomEffect`

Table 7-2: X11/twm Profile Options Related to Features

7.3.2.5. Model Validation

The feature model may be used to predict behavior in a given scenario based on the feature values of a specific system. The results of having two (or more) specific systems perform an operation may be compared with the results predicted by the feature model instantiations for those systems. Any variation between the predicted and actual results should indicate problems with the system descriptions of one or both systems.

An example is a comparison of *X10/uwm* and *SunView* in performing a *move* operation on an existing window at the bottom of the stacking order, and trying to push it off the screen while it is overlapped by another window. While the scenario is simple, it can involve a significant number of features. The results of the scenario (given by listing features from the catalogue for each system) are as follows:

X10/uwm:

- `actionObject`: Command is selected before the window.
- `opaqueFeedback`: Feedback is an image of the window itself.
- `partiallyOffScreen`: Window moves partially off the screen.
- `exposeAfterMove`: Window is exposed after the move.
- `realEstateMode`: Window becomes the input focus after move due to the pointer position being above the window.

SunView:

- `objectAction:` Window is selected before the command.
- `ghostFeedback:` Feedback is an outline of the window.
- no `partiallyOffScreen:` Window kept inside screen.
- no `exposeAfterMove:` Window still partly hidden.
- `listenerMode:` Window becomes the input focus before the move by clicking to select it.

Due to the choices made by the designers of X10/uwm and SunView, and the runtime selections made in this example for X10/uwm (`opaqueFeedback` and `realEstateMode`), the results are direct opposites of one another in terms of the possible feature values.

7.3.2.6. Automated Tool Support for Features

Manually creating a feature model that correctly describes a complex domain is a large effort; validating that model in some way is still more difficult. As part of the feasibility study for performing useful, "real-world" domain analyses it became clear that manual methods would not suffice, even in a relatively small example. Because the FODA method is new, and no existing automated tool support was available, a prototype tool was developed using Prolog. The primary function of the tool is to validate the usefulness of the feature analysis approach, and secondarily to establish some baseline requirements for future automated support for the method.

The tool is separate from the information about the domain being analyzed, so that it may be applied to any domain. The features are stored in a Prolog fact base, along with the composition rules and other related information. The tool supports definition of existing or proposed systems by allowing arbitrary sets of feature values to be specified and checked. The composition rules relating the features are enforced, as are standard rules about completeness of the model.

Given a set of user-specified (i.e., "marked") features, the automated features tool presently performs the following functions:

- Checks for *all* features that are specified, but which may not be *reachable.*
- Marks a feature as "valid" if it is either:
    - marked "valid",
    - mandatory,
    - *not* marked "invalid", or
    - *required* by a "valid" feature.
- Marks a feature as "invalid" if it is mutually exclusive with a "valid" feature.
- Produces an error if a feature is marked as both "valid" and "invalid."
- Enforces the proper selection of alternatives:
    - at least one alternative *must* be marked "valid."

- more than one alternative *cannot* be "valid."

The features stored in the fact base have six pieces of information[14] attached to them (see Appendix A). These are illustrated with the example of the `exposeAfterMove` feature below.

Name:	`exposeAfterMove`
Description:	*Expose the window at the end of a move operation*
Type:	*optional*
Parent:	`moveWindowOp`
Rules:	requires `overlappedLayout`
Source:	*SunView window system experience*

The fact as it is stored in Prolog format is as follows:

```
daFeature(exposeAfterMove,
   'Expose the window at the end of a move operation.',
   optional,
   moveWindowOp,
   [requires(overlappedLayout)],
   'SunView window system experience').
```

In addition to the general feature model, specific systems (i.e., sets of feature values) may also be stored; as they are developed they may be periodically checked for consistency. To illustrate this consistency checking process, the following small example of an inconsistent system description will suffice.

An imaginary (and certainly incomplete) system is specified with a set of only three feature values:

- `zapEffect`
- `eraseBefore`
- `opaqueFeedback`

While the actual automated features tool will report in detail the incompleteness of the model, for now the focus is only on the consistency of the feature model or lack thereof. It is already clear from the discussion of the sample composition rules in Section 7.3.2.2 that these particular features are not compatible, and in fact a run of the features tool on the above "system" definition produces a report including the following messages:

- `ghostFeedback` is selected due to `zapEffect`.
- `eraseAfter` is selected due to `zapEffect`.
- `moveErasure` is selected due to `ghostFeedback`.

[14]The issue and decision information should be stored in the Prolog automated features tool with the feature, but currently is not.

- `moveErasure` is *invalid* due to `opaqueFeedback`.
- More than one alternative of `interactiveFeedback` has been selected- only one is allowed.

In this case the features tool attempted to make the description complete where possible, without knowing that doing so would lead to inconsistencies. For example, it added `ghostFeedback` and `eraseAfter`, as well as others. There are several inconsistencies here: both `ghostFeedback` and `opaqueFeedback` have been selected (i.e., more than one alternative); both `eraseBefore` and `eraseAfter` have been selected; `moveErasure` has been marked as both valid and invalid. While the system will continue to discover other errors, only one is necessary to point out that the system description is incorrect. As is the case with many similar systems (notably compilers), further messages may provide little additional information.

7.3.3. Functional Model

The functional model of the domain analysis identifies functional commonalities of the applications in a domain. The model also seeks to identify and compare differences between these related applications. The model abstracts and represents these common/differing functions so that a specific application can be viewed as an adaptation or *refinement* of the model.

The development of the functional model depends on the features and entity-relationship models. A high-level, abstract functional model is derived from the common features and entities of these models. Features from the feature model parameterize the functional model through refinement by representing alternative and optional functions. While the feature model is used to communicate between the requirements analyst and the user (see Figure 3-4), the functional model together with the feature model support communication between the analyst and the software designer. The user's choice of features provides actual values for the parameters of the functional model.

The example domain analysis uses Statemate Statecharts and Activitycharts to represent the functional model [Harel 89]. As was the case in the Prolog automated features tool, no existing tool adequately handles the requirements necessary for modelling common functionality and handling parameterization through features. Statemate offers a good, general-purpose specification and documentation tool, though the application of the tool to support domain analysis requires tailoring. Through tailoring to handle domain analysis, Statemate can:

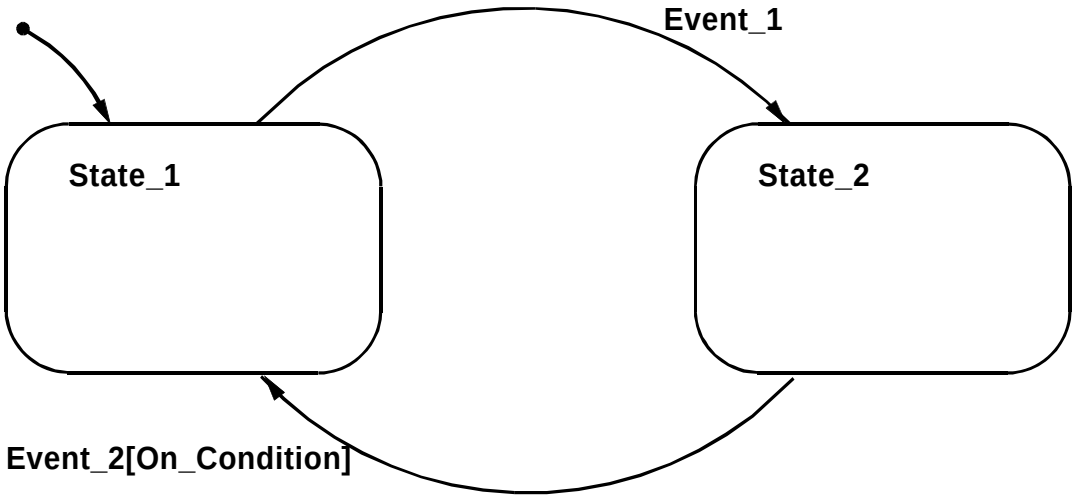
- Capture commonality
    - Statecharts show all states and transitions for specifying a behavioral view.
    - Activitycharts show common functions and data flows (input and output) for specifying a functional view.
- Parameterize differences through features

- Statecharts show alternative/optional features as conditions for modifying behavior.
- Activitycharts show optional data flows and provide textual descriptions.

While Statemate can support many aspects of functional modelling, it does have some weaknesses (presented in Chapter 8). In addition to its use as a modelling tool, the experience of applying Statemate to the functional modelling task can provide guidance in establishing specific requirements for a domain analysis support tool, as is discussed in Sections 8.1.3 and 8.2.

7.3.3.1. Specification of Behavior – State Transition View

The behavioral view of a system can be specified by characterizing the system in terms of *states* and *state transitions*. A state represents a conceptual mode of a system. For the window manager domain, examples of the states include window creation, operation selection, and window moving. A transition causes transfer from one state to the next in response to an *event*, such as the window manager user's selecting an operation to perform on a window, or the user's deleting a window. In addition, transitions can be affected by *conditions*; the value of the *condition* will determine whether the transition will take place. The Statemate tool captures this behavior in a Statechart, as shown in Figure 7-8.



(In this example, the arrow into State_1 indicates that the system starts in State_1. A transition to State_2 will occur when Event_1 takes place. The system will transition back to State_1 when Event_2 occurs *and* the boolean On_Condition is true.)

Figure 7-8: Statechart Illustrating Behavioral View of a System

A domain analysis of window managers must capture the behavior exhibited by *all* of the systems studied in the domain analysis and represent that behavior. In the window manager domain analysis features that are common to all window managers appear on

Statecharts as mandatory states. Alternative and optional features are handled through conditional transitions and optional states.

For the window manager domain, the behavior includes information on the state of *windows* controlled by the window manager and on the state of actions of the *user* of the window manager. Figure 7-9 shows the parallelism of these two views of the behavior of systems in the domain. The two parallel states show:

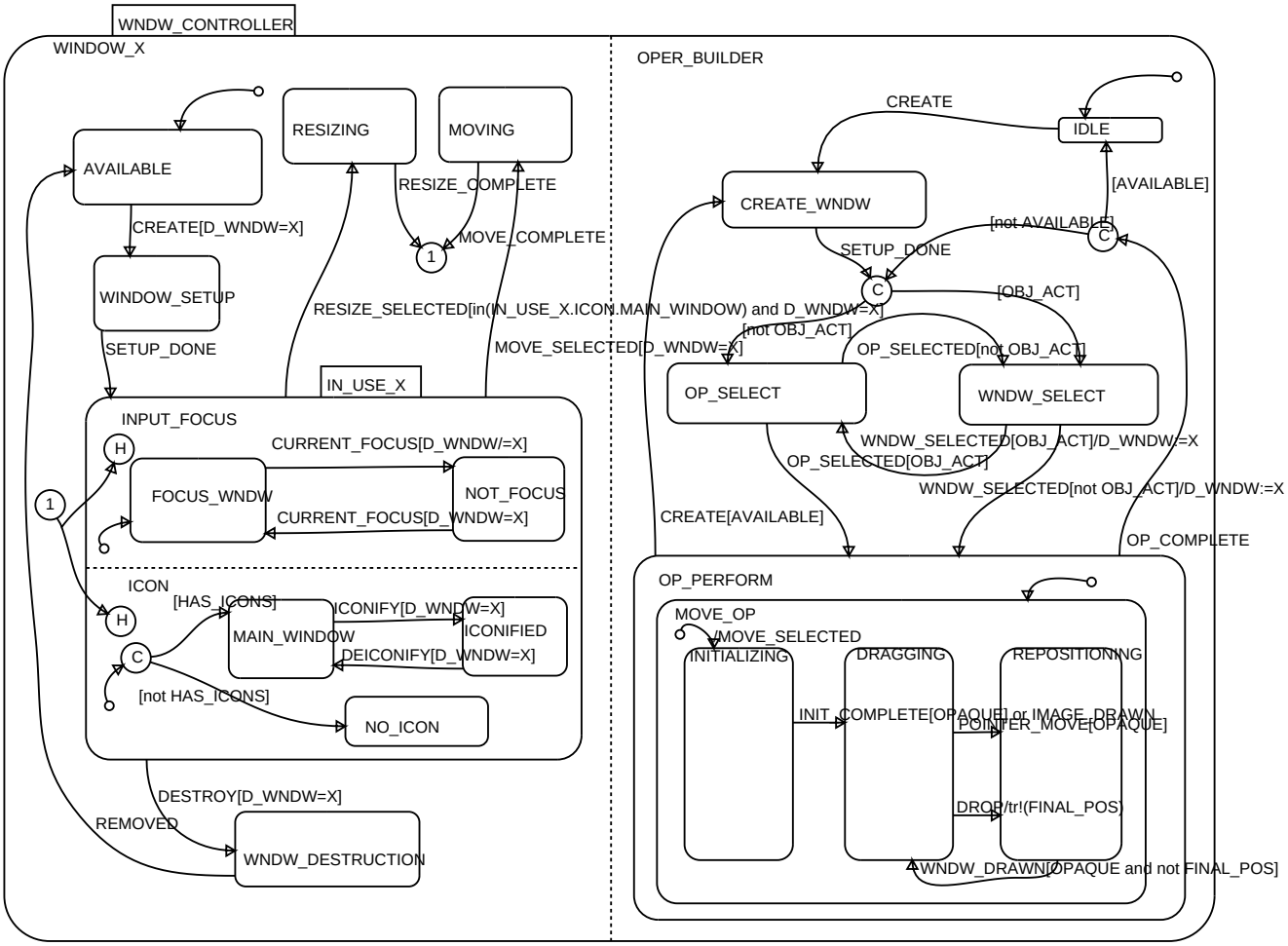


Figure 7-9: Statechart Illustrating Behavioral View of Window Manager

1. *WINDOW_X* - the behavior of the system in response to user operations. This side of the state chart shows the state of a given window on which the user may choose to operate.
2. *OPER_BUILDER* - the behavior of the system to support user operations. Once a user creates a window, he can perform operations on that window.

The functional analysis focussed again on the *move* operation to illustrate the application of

the method. Within the state called WINDOW_X are substates in which a window operates. The states labeled AVAILABLE and WINDOW_SETUP are the window states that exist before and during creation of the window, respectively. Following window creation, the window goes into an IN_USE_X state. The IN_USE_X state illustrates the manner in which a feature can parameterize the functional model. When the window enters IN_USE_X, it is always in state FOCUS_WNDW indicating it is the *input_focus*. This mandatory feature is common to all window managers. However, the transition to MAIN_WINDOW is conditional; this *optional* state is parameterized via the *hasIcons* condition. This parameter is derived directly from the feature model; the `has_icons` feature is optional among window managers and parameterizes the functional model.

The state transitions (events) within the WINDOW_X behavioral view of the window manager are also derived from features of the domain. For example, each state transition under the WINDOW_X state occurs in response to an operational feature (e.g., *create*, *destroy*, *move*, *resize*). Table 7-3 illustrates this correspondence. The features shown in the table from the feature model are represented as events prompting a state transition in the functional model.

Feature	Event
create	create
destroy	destroy
move	move_selected
resize	resize_selected
iconify	iconify

Table 7-3: Features and State Transitions

The states under OPER_BUILDER are also related to specific features of a user's interactions with window managers. The basic operational cycle shown in this state consists of:

1. Creating a window.
2. Selecting a new operation (which may be to perform an operation on that window, perform an operation on another window, or create a new window).
3. Performing the operation.

The feature model gives explicit guidance in the specification of the user's interactions. For each state transition that a window may undergo in WINDOW_X there is a corresponding transition in the user's interaction with the window manager shown in state OPER_BUILDER.[15] For example:

Creating a window

- WINDOW_X: The *create* event causes a transition to the WINDOW_SETUP state to perform the create operation. When

[15]For clarity, the Statechart does not show these transitions to the level of detail necessary to explicitly show each transition. Only those for creating and moving a window are shown.

this operation is complete, the *setup_done* causes a transition to IN_USE_X.

- OPER_BUILDER: When the user performs a create operation the *create* event causes a transition to the CREATE_WNDW state. From this state, the *setup_done* will cause the transition out of the CREATE_WNDW state.

Moving a window

- WINDOW_X: The *move_selected* event causes a transition to the MOVING state. When the user completes the move, the *move_complete* event causes a transition back to the state IN_USE_X (signified by the connector labelled "1" in Figure 7-9).
- OPER_BUILDER: The *move_selected* event causes a transition to the INITIALIZING state within MOVE_OP. The states subsequent to INITIALIZING perform the move operation. The *move_complete* event (shown in detail in Figure 7-11) causes a transition out of the MOVE_OP state.

The feature model also provides alternative and optional states for user interaction with the window manager. One feature of all window managers is the order in which a user selects a window and an operation to perform on that window. The alternatives are:

- `objectAction`: The user first selects the window, then chooses an operation.
- `actionObject`: The user selects an operation, then designates the window.

The conditional transition following the *setup_done* event in Figure 7-9 demonstrates the use of these features to parameterize the functional model. The *obj_act* condition will cause a transition to the WNDW_SELECT state first, in accordance with the `objectAction` feature. After selecting the window, the feature is completed with the transition to the OP_SELECT state to allow the user to select the operation. When `actionObject` is the feature, the *not obj_act* conditional causes the states to be OP_SELECT first, then WNDW_SELECT in accordance with `actionObject`.

A detailed study of the common features of the *move* operation (Figure 7-10) will provide a further understanding of the application of the FODA method in establishing the functional model. This example will also show the interaction between the behavioral and functional aspects of the model. The basic *move* operation consists of three successive states, common to all window managers:

1. *Initializing:* obtaining current window position and other parameters and constraints
2. *Dragging:* moving the window or a ghost image of the window to an interim position
3. *Repositioning:* redrawing the entire window and establishing the new location of the window, both visually on the screen and in internal tables.

Transitions between these states occur in response to specific events, such as the comple-

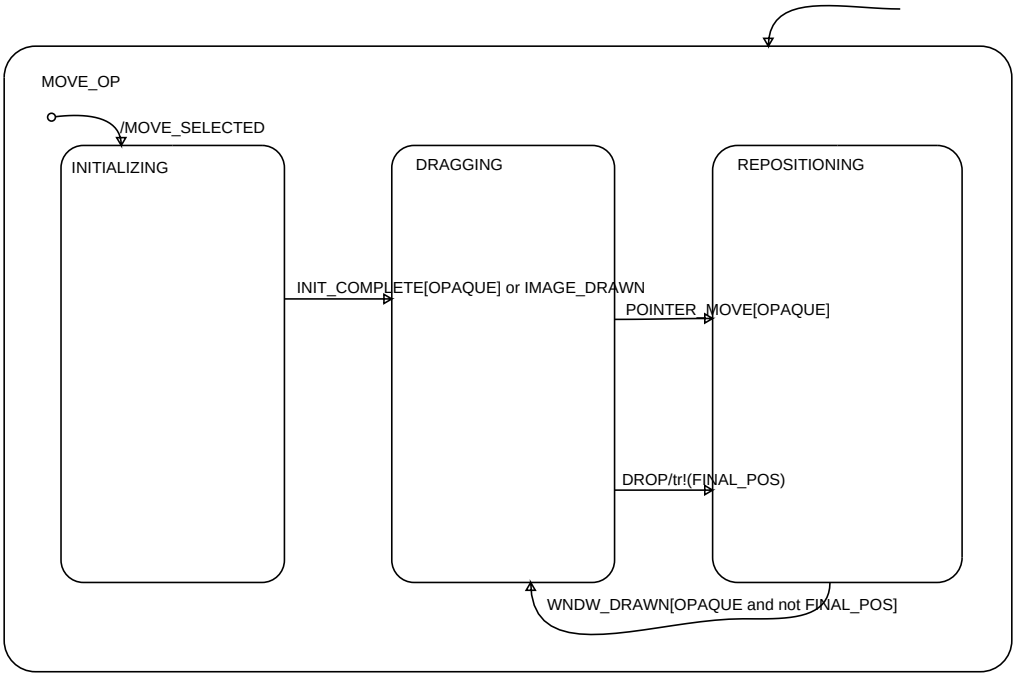


Figure 7-10: Statechart Illustrating Basic Move Behavior

tion of initialization (event *init_complete*) or the movement of the pointer (event *pointer_move*).

One significant difference between window manager move operations is highlighted by the parameterization for the `opaqueFeedback` vs. `ghostFeedback` features.

- Under `opaqueFeedback`, each movement of the pointer causes the entire window to be redrawn. The behavioral view must show this feature as causing a loop from dragging to repositioning and back to dragging. This loop causes the window to be redrawn with each pointer movement as shown by the transition from DRAGGING to REPOSITIONING that occurs on event *pointer_move* when the *opaque* condition is true.
- For `ghostFeedback`, the looping occurs entirely within the dragging state – the window is not redrawn until it is dropped via event *drop* in its final desired new location.

The combination of other *move*-related features leads to a more detailed view of the *move* operation in Figure 7-11. This *refinement* of the more abstract functional view of Figure 7-10 shows the effect of both alternative and optional features. These features become Statemate *conditionals* in the figure within each of the higher level states. The following list shows the result that selecting certain features has on transitions:

- INITIALIZING: The `opaqueFeedback` feature is parameterized as the *opaque* condition, which is mutually exclusive with the ghost condition. This condition

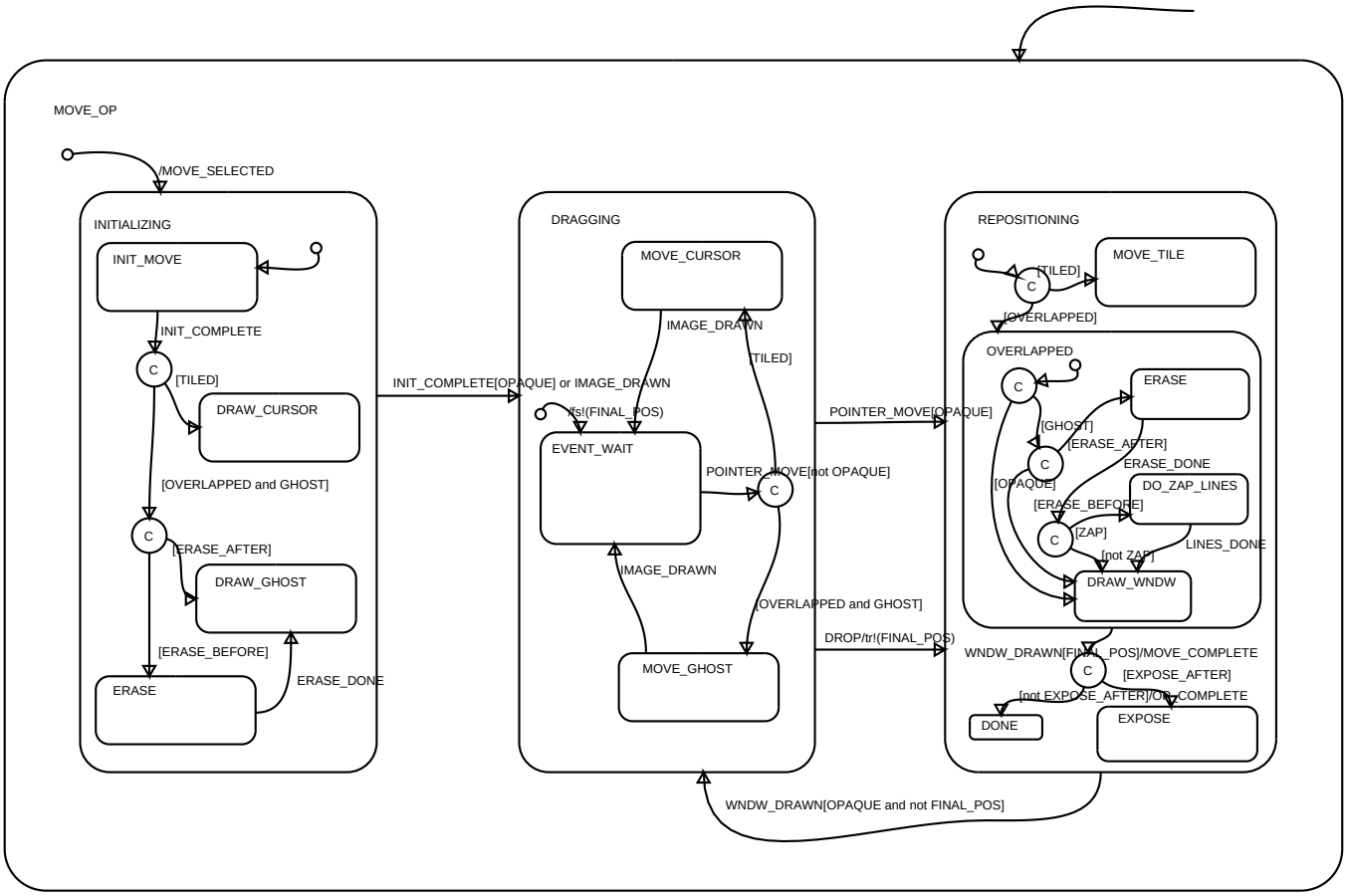


Figure 7-11: Statechart Illustrating Details of Move Behavior

causes a transition from the INITIALIZING state when the event *init_complete* occurs. In order to support the feature, this parameterization causes the window manager to bypass other substates in INITIALIZING that erase the window or draw a ghost.

- INITIALIZING: The `eraseBefore` and `eraseAfter` features are also parameterized. The condition *erase_before* causes a transition to erase the window before drawing a ghost; *erase_after* bypasses this state.
- DRAGGING: The `opaqueFeedback` and `ghostFeedback` features parameterize this state. The condition *not opaque* (i.e., ghost) will support the `ghostFeedback` feature, causing an internal loop to drag the ghost image with each pointer movement. The *opaque* condition causes a transition to the REPOSITIONING state with each *pointer_move* event, causing the window to be completely redrawn in a new position every time the pointer is moved.
- REPOSITIONING: The `zapEffect` feature parameterizes the OVERLAPPED substate within REPOSITIONING. The condition *zap* will support this feature by causing a transition to DO_ZAP_LINES. When the `zapEffect` condition is not true, for example when the `opaqueFeedback` feature is in effect due to the composition rule relating the two discussed in Section 7.3.2.2, the transition will

be to the DRAW_WNDW state, to redraw the window in a new position without zap lines.

This detailed functional model can be instantiated for any valid combination of features to *specialize* the model for a particular window manager. This type of specialization will be presented in Section 7.3.3.3.

7.3.3.2. Specification of Function – Data Flow View

The functional view of the functional model is based on data flow techniques. Among the factors that this view establishes are:

- functions of applications in a domain
- their inputs and outputs
- internal data and data structures
- data flow between functions

As with the behavioral view, the features and entity relationship models play a significant role in the functional view of the domain. The features can lead to mandatory, optional, and alternative functions or data flows:

> The `constrainedMove` feature will determine whether window motion can be restricted in the horizontal or vertical direction during a *move*. This will affect the functionality of converting new pointer position to new window position.

Features will also be a factor in characterizing the data inputs and outputs:

> The `windowShape` feature will parameterize the data flow to a function to draw a window; a window manager that supports non-rectangular windows will have a data flow different from that of a window manager supporting only rectangular windows. The functional view must be general enough to accommodate either value, rectangular or non-rectangular, for this feature.

The entity-relationship model and attributes provide guidance in specifying the data structures for the functional view:

> The *window geometry* attribute defines the window size, shape, position, and, if applicable, stacking order position. The functional view of the window manager must capture this information in its internal data structures.

For the window manager domain, the functional view includes functions such as setting the position of the pointer, drawing a new window, and erasing a window. These functions must input data to obtain the positions of windows and the pointer, window shapes and sizes, and other parameters and constraints. The functional view also defines the relationship between functions:

> Drawing a window requires input from the pointer position to determine the new window position and from the window data to determine the window's size and shape; the function must output the new window position to a window data store.

The Statemate tool handles these functional specification requirements through the Ac-

tivitychart. Boxes on the activity chart shown in Figure 7-12 represent functions, while flow lines represent data flows. In addition, the activities are controlled by a *control activity*, namely the WNDW_CONTROLLER shown in Figure 7-12 and in detail in Figure 7-9. During each state of Figure 7-9, the system performs one or more specific activities. The mandatory, alternative, and optional features of the behavioral view will control exactly which activities are performed for a specific window manager. This *separation of concerns* between control (behavioral view) and function (functional view) is directly supported by Statemate.

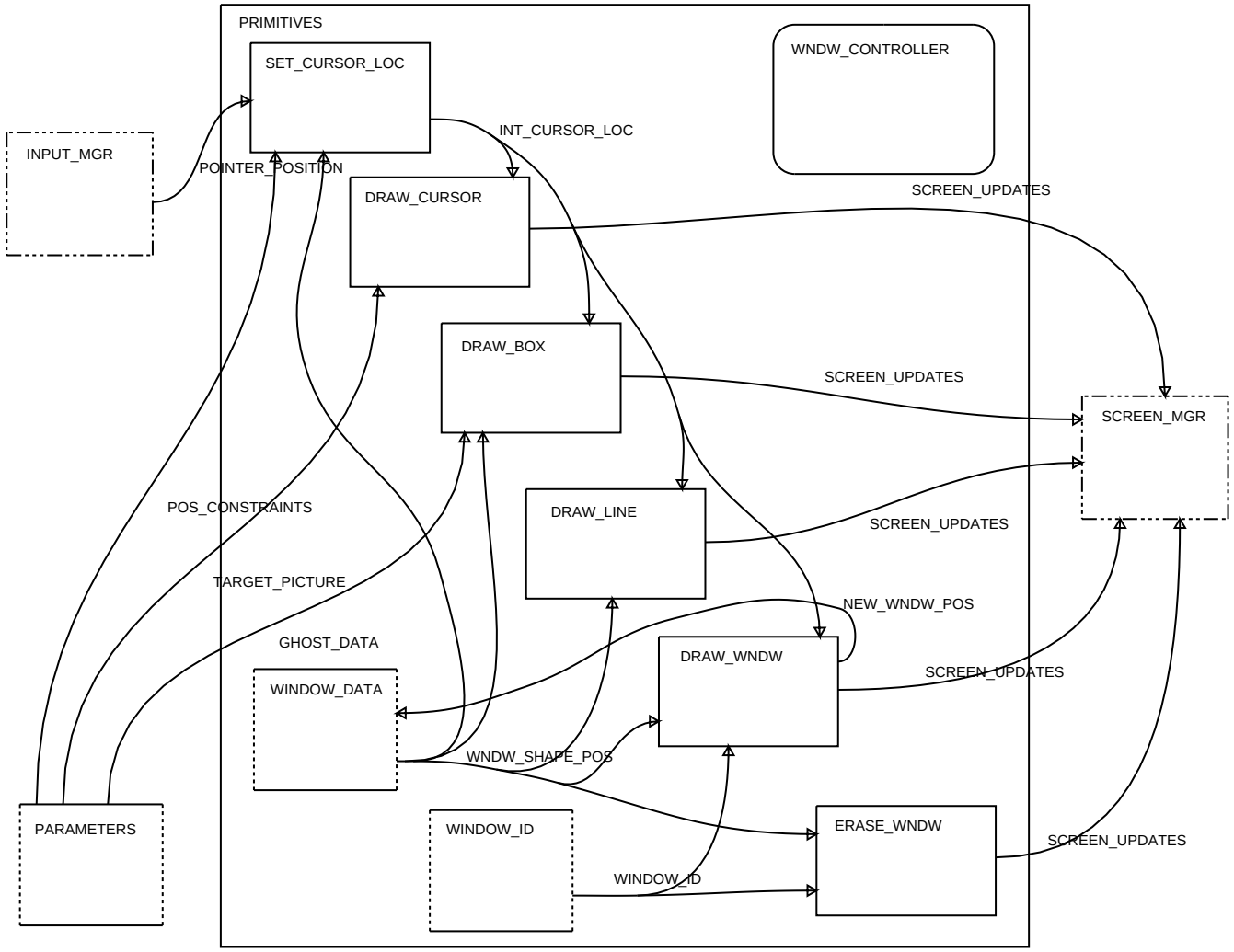


Figure 7-12: Activitychart Illustrating Functional Specification

The sequence of activities performed by several window manager states will illustrate the purpose of the Activitychart. During the MOVE_GHOST state of DRAGGING in the *move* operation (Figure 7-11), the window manager must determine the new position of the ghost as well as its shape and size. Figure 7-13 shows the activities and data flows for this state, as a subset of those of Figure 7-12.

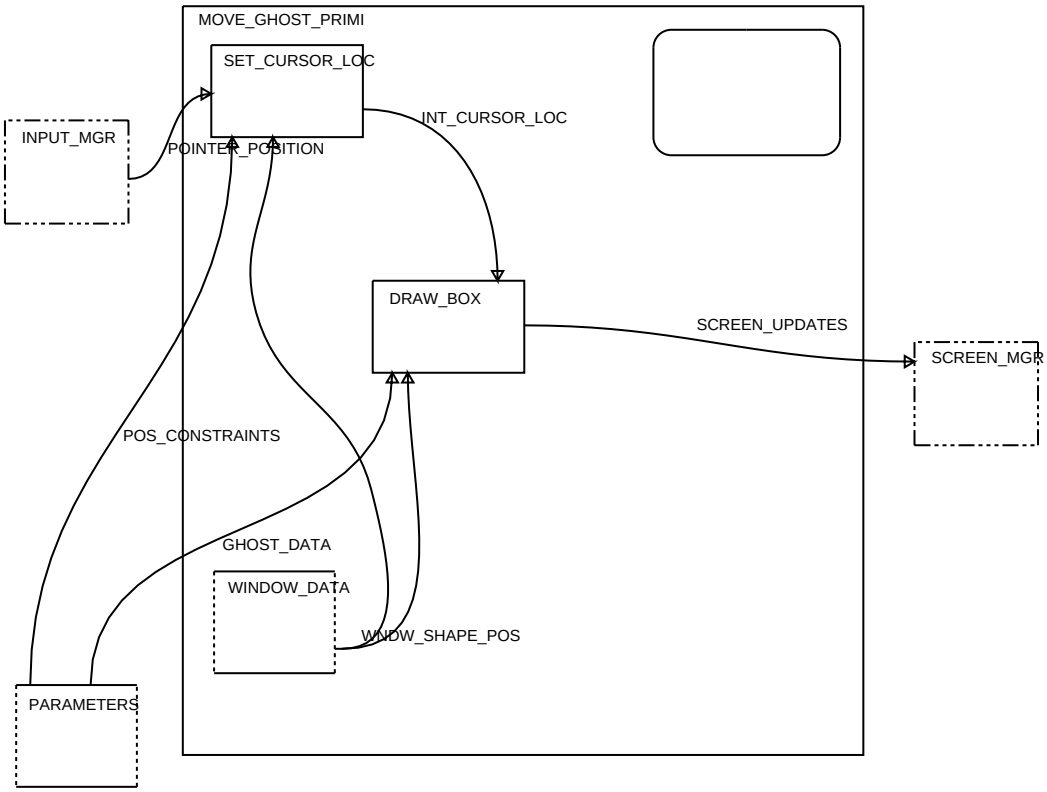


Figure 7-13: Activitychart Illustrating MOVE_GHOST Activities

The *set_cursor_loc* activity will get the pointer location from the input manager and position constraints from the external *parameters* data store. The activity takes the actual location of the pointer (labelled *pointer_position*) and converts it to an internal location. The conversion is necessary to account for features such as `constrainedMove` or `partiallyOffScreen`, which restrict the movement of the window and require reinterpretation of pointer position. The data flow labelled *int_cursor_loc* represents this position value for the location. The *draw_box* activity will then draw a ghost image, using ghost data (such as outline and background characteristics) and data on the shape and location of the window being moved. The *draw_box* activity outputs data to update the screen display through the *screen_manager* activity.

During the DRAW_WNDW state of REPOSITIONING in the *move* operation (Figure 7-11), a similar set of activities will occur. Figure 7-14 shows the activities and data flows for this state. As with the MOVE_GHOST state, the *set_cursor_loc* activity must be parameterized

for the `constrainedMove` and `partiallyOffScreenWindows` features of the *move* operation. The *draw_wndw* activity inputs *wndw_shape_pos* to get the window characteristics and outputs data to update the screen. In addition, the activity updates the window geometry; these updated values are output as *new_wndw_pos* to the *window_data* store.

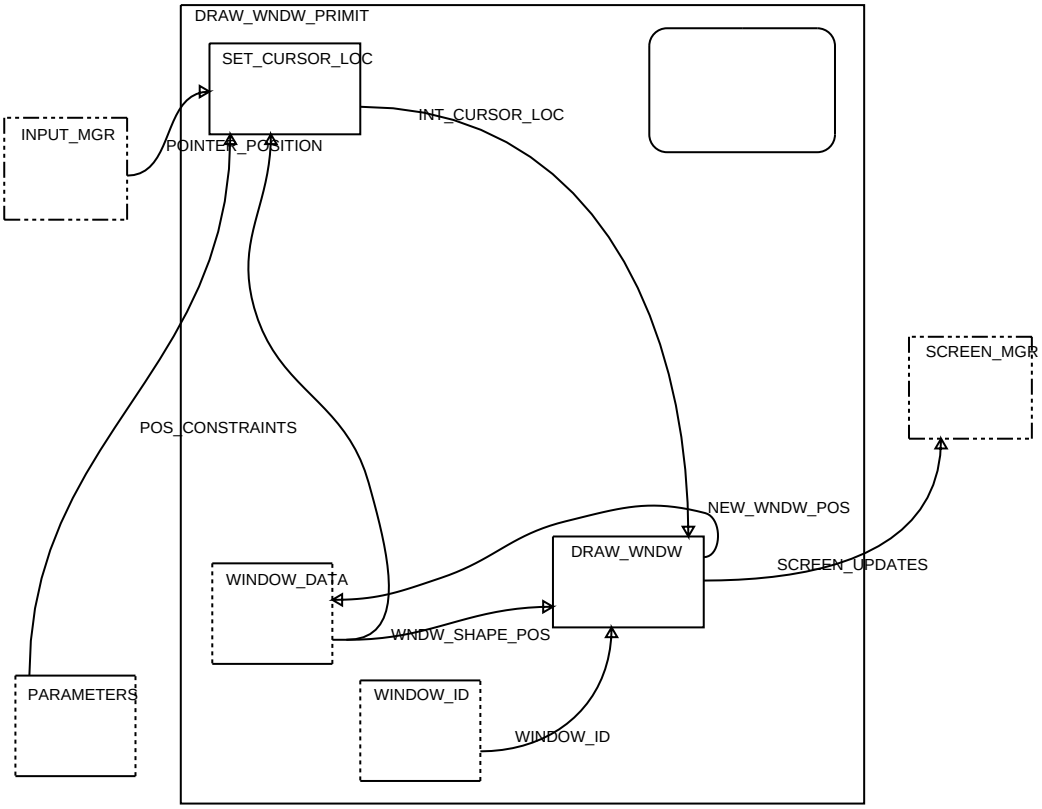


Figure 7-14: Activitychart Illustrating DRAW_WNDW Activities

7.3.3.3. Validation of Functional Model

A critical aspect of the domain analysis is verifying that the model can be used to represent the performance of a new or existing system. The verification of the feature model was described in Section 7.3.2.5. To perform this verification, the domain model was *refined* using features of two different window managers. In verifying the functional model, these refinements entail parameterization through setting conditions to account for differences in the behavioral and functional views of the two managers. The use of Statemate to represent the functional model provides an additional benefit; the performance of the functional model can be *simulated* in Statemate to test if the parameterization corresponds to expected operations.

For each of the two systems in the comparison, *X10/uwm* and *SunView*, the group of relevant features was set via conditional parameters. Three alternative features were examined for the purpose of this validation:

X10/uwm	SunView
`objectAction`	`actionObject`
`opaqueFeedback`	`ghostFeedback`
`exposeAfterMove`	no `exposeAfterMove`

Running the simulation using the features of the X10/uwm window manager produced the results shown in Figure 7-15. After the INIT_MOVE state, the operation immediately transitions to the DRAGGING state because there is no change to the window under `opaqueFeedback` during INITIALIZING. When a window is being dragged under the X10/uwm *move* operation with `opaqueFeedback` selected, each pointer movement causes the entire window to be redrawn in the new position. Statemate simulates this performance by making a transition from the DRAGGING state to the REPOSITIONING state with each *pointer_move* event. The simulation continues to model the performance of the X10/uwm window manager by looping back to the DRAGGING state following the redrawing of the window in the DRAW_WNDW state, until the user indicates the final position of the window via the *final_pos* condition on the *drop* event. (In most window managers, this event is caused by the release of a mouse button or similar user input.) Following the move, X10/uwm *exposes* the window, i.e., moves it to the top of the stacking order, in the new position. The simulation makes a transition to the EXPOSE state to perform this operation.

Running the simulation using the features of the SunView window manager produced the results shown in Figure 7-16. Because the SunView move operation supports only `ghostFeedback`, after the INIT_MOVE state the manager makes a transition to the DRAW_GHOST state, as is the case when the SunView window manager is performing a ghost move. The *erase_after* condition is also followed to indicate that the window will not be erased until the move is complete. Within the DRAGGING state, each movement of the pointer causes a corresponding ghost movement. The loop states of EVENT_WAIT and MOVE_GHOST simulate the ghost movement following a pointer movement within DRAGGING. When the user signals the final position for the move, a *drop* event occurs. This causes the transition to REPOSITIONING. During this state, the SunView window manager erases the old window and redraws it in the new position. Statemate successfully simulated the performance of both actual window managers.

This discussion has traced only the *state transitions* of the *move* operation. The actual simulation also shows the *activities* and *data flows* performed in response to *states* and *transitions*.

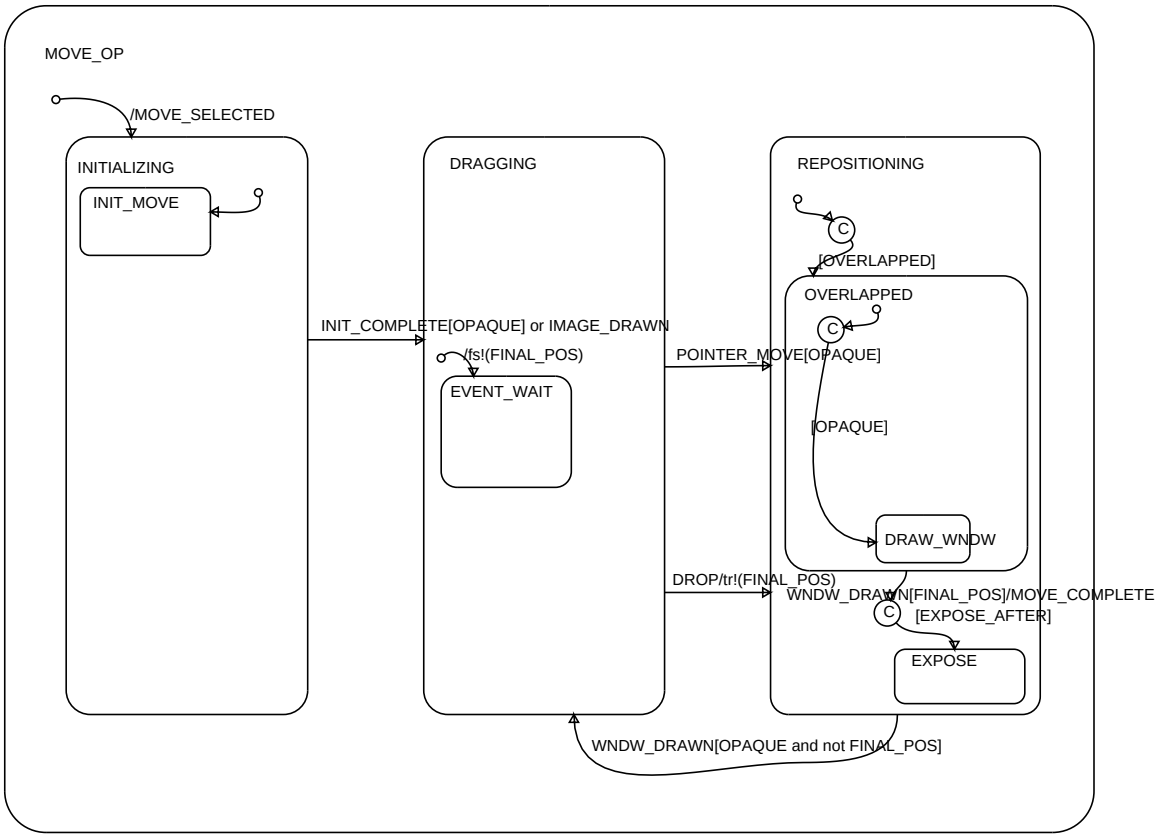


Figure 7-15: Statechart Illustrating X10/uwm Specification

7.3.3.4. Application of Functional Model

The functional model supports several important factors in performing the domain analysis:

- It captures commonality of data and control flow.
- Through applying Statemate, it verifies system performance through simulation.
- It provides requirements for architectures and reusable components.

The example analysis did not include the analysis of architectures or components. The feasibility study did produce a functional model that can be applied during architecture modelling:

- Behavioral view: used to establish control of the system in the process interaction model.
- Functional view: used to establish packaging of functions and data objects in the module structure charts.

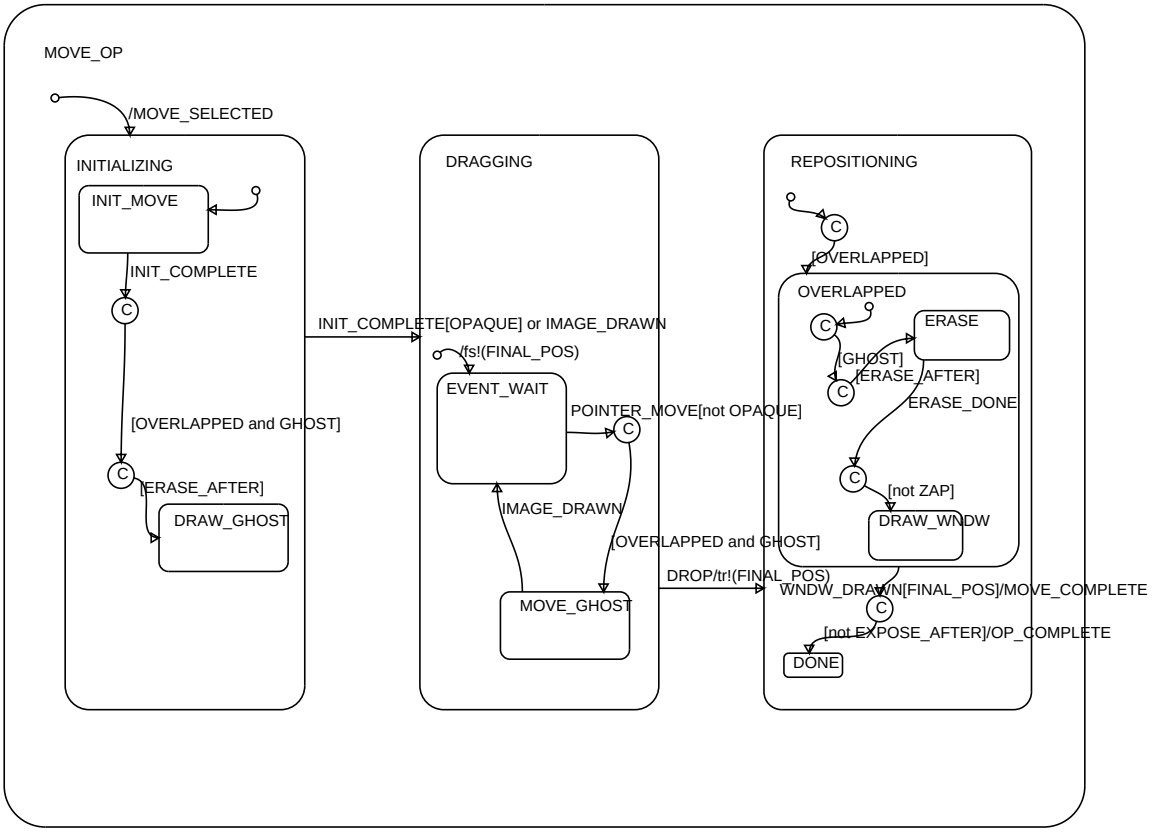


Figure 7-16: Statechart Illustrating SunView Specification

8. Discussion of the FODA Method

The primary purpose of the sample analysis was to apply the FODA method to a realistic domain. Although the sample analysis did not cover the entire method (omitting the architecture modelling phase), the application of the method uncovered several limitations which are discussed in the next section. The sample analysis also pointed out the need for follow-on study to broaden the application of the method and to explore new uses for the domain products.

8.1. Limitations of the FODA Method

8.1.1. Methods for Representing Generalization/Specialization

Although a set of modelling primitives for capturing commonalities and differences of a domain is identified in the FODA method, the method does not provide representation techniques or tools that will adequately support these primitives. For instance, generalization/specialization relationships cannot be represented graphically at the functional level and the method must rely on textual description. Also, parameterization of the functional model is not adequately supported. Statemate *conditions* for feature parameterization could not be distinguished from other *conditions* used in the specification. A representation technique that will support these modelling primitives and a support tool that will allow the users to instantiate diagrams easily are necessary.

8.1.2. Composition Rules vs. And-Or of Features

The hierarchical relationship between the features in the and/or tree is an alternate graphical representation of the *requires* composition rule (discussed in Section 7.3.2.2). While using two alternate representations of the same relationship complicates the feature model, the feature diagram provides a way for users of the model to "see" some of the feature relationships. That is difficult to do with a model composed solely of composition rules. Sophisticated automated support for the interactive display of the feature model, such as hypertext techniques, could provide displays that would make the feature diagram redundant and hence unnecessary.

8.1.3. Manual vs. Automated Methods

The domain analysis of window managers was first approached with the intention of performing the analysis using only manual techniques. As the amount of information needed to describe the domain grew, the manual techniques became more complex. For example, the feature diagram had no way of displaying the effects of the composition rules on the relations between features to show that the existence of one feature was conditional on another feature. To handle this some notational extensions to the diagram were tried, but these only made an already complex diagram larger and more abstruse. The feature diagram had been split across several pages along arbitrary boundaries in an attempt to make it more manageable, and contained inconsistencies that could be found only through exhaustive

manual examination. This situation was a primary reason for building a prototype automated features tool to represent the feature model and support consistency and completeness checking. The manual methods were insufficient to handle a set of little more than 100 features, and would clearly be inadequate to the task of modelling a larger domain.

Representing the results of a domain analysis process is primarily a task of representing a large amount of knowledge, and providing facilities so that the user can access that knowledge quickly and easily. The goal of domain analysis tool support is to offer an integrated environment for collecting and retrieving the domain model and architectures. The current set of manual and independent semi-automated methods does not meet this goal. It has, however, clearly pointed out the problem which must be addressed, and provides a basis for working on the problem.

The sample analysis used Statemate to prototype a tool that could take a consistent set of features produced by the automated features tool and support the functional modelling of the domain. Statemate has provided valuable support to the domain analysis, but also has several limitations. Among the strengths of the Statemate tool are:

- It is a good analysis tool.
- It supports parameterization.
- It is a production-quality tool.

Some of the weaknesses of the Statemate tool are:

- It is not scalable without tailoring.
- The transitions/conditionals hide commonality.
- It is difficult to transition due to the cost and training required.

Statemate can also serve as a means of establishing requirements for domain analysis tools as discussed in Section 8.2.

8.2. Future Directions

8.2.1. Near-Term

8.2.1.1. Formalization of Features

The FODA method does not apply formal techniques in the specification of features. The specification of features is made informally in English text, which can result in ambiguity and inconsistency. For example, features from the window manager domain such as `constrainedMove` and `zapEffect` could have been specified more precisely using a formal specification technique. The project applied an algebraic specification technique to specify the concept of stacking order in the window manager domain.[16] However, this effort was not expanded to cover the entire model.

[16]This technique is not discussed in this report.

8.2.1.2. Formalization of Issues/Decisions

The FODA method records *issues* related to each decision point of the feature model to help users make selections of both optional and alternative features. However, it does not provide a model showing how these issues are related. For example, in the case of automobile features in Figure 5-1, the *issue* of *operating cost* is related to *maintenance cost* and *fuel efficiency,* and to help users make decisions this relationship must be captured in the model with quantitative values. (One way of implementing this feature is to define issues as feature attributes; the issue model would be used to define relationships between these attributes, i.e., issues.)

Also, a customer often has a set of conflicting issues. For example, if one wants to buy a car with good acceleration and low fuel consumption, he identifies the requirements (i.e., issues) that are contradictory and a compromise decision must be made. While this type of situation is resolved on a regular basis without automated support, more complex sets of conflicting issues are not so easily decided. Automated assistance is needed to identify these conflicting issues and resolve them to make an optimal decision.

8.2.1.3. Tool Support for Domain Analysis

The feasibility study determined the need for tools to support both the process of domain analysis and the process by which the products of domain analysis support software development. These tools are needed to deal with the volume and complexity of information gathered during the domain analysis and the presentation of that information in specific domain analysis products. Tools are also required to provide a user with an understanding of the domain and to support the user/implementor interaction in developing a new system within the domain. Domain analysis tools will also be used to support the development and application of reusable software.

The study established four levels of tool support, ranging from manual methods to those that are specifically intended to support domain analysis. Figure 8-1 illustrates the successive levels of support and provides examples of each. The first level provides only a database of information that the user must handle through manual means to derive any new results. The second level takes the data from the first level and automates part of the derivation process. For example, the automated features tool (see Section 7.3.2.6) checks consistency between features and the Statemate-supported simulation of the functional model. The third level, that of integrated tools, takes general-purpose tools that may be used in an informal way at the second level and incorporates their use into the method. Tools at the fourth level are those specifically developed or tailored to support the specific needs of domain analysis.

One task in future domain analysis methods investigation will be to further integrate tool support into the domain analysis method and produce requirements for specific domain analysis support tools. This investigation will determine which of the four approaches provides the best support for the FODA method and which can be most easily transitioned. The investigation will continue to work in well established domains to refine these requirements in a prototype tool.

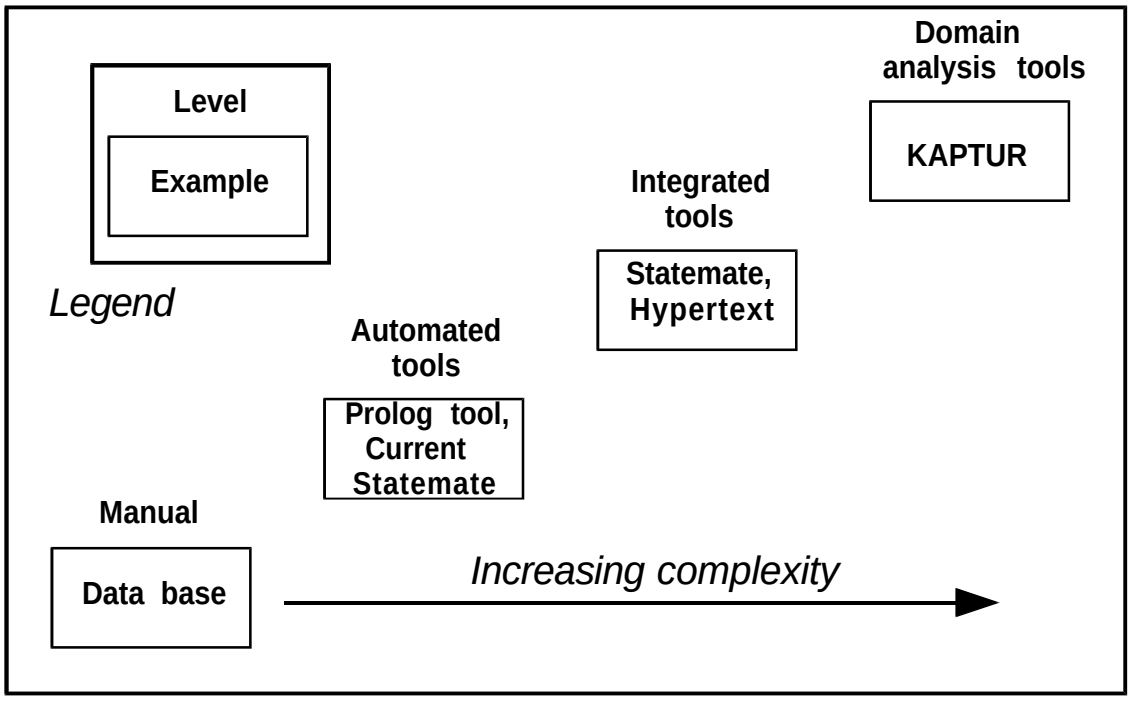


Figure 8-1: Levels of Domain Analysis Tool Support

8.2.1.4. Handling the Feature Binding Time Attribute

One of the fundamental trade-offs that a system designer makes is to select binding times for features, which can then have a significant impact on system functionality, size, speed, and architecture. As discussed in Section 7.3.2.4, some window managers have already incorporated a range of variation into the executable, allowing activation-time and runtime tailoring of functionality. It is worth generalizing this concept into another attribute of a feature, *feature binding time.* For example, in describing the features of a specific system it is necessary to select all of the features and feature values which that system has, which often includes selecting a single feature from a set of alternatives. In the case of an activation-time tailorable system such as X11/uwm (using a profile file) there may be no single appropriate alternative. Section 5.1.2 describes how the model must designate at what time the alternative will be bound: *compile-time, activation-time,* or *runtime*. An example of a traditional *compile-time* feature might be `tiledLayout` versus `overlappedLayout` (where the constraints are "hard-coded" into the window manager source), an *activation-time* feature might be `ghostFeedback` versus `opaqueFeedback`, and a *runtime* feature might be `realEstateMode` versus `listenerMode` (in X10/uwm both options are available simultaneously).

In addition to affecting the feature model, the ability of the user to postpone the selection of features until activation-time or runtime can significantly change the software architecture by

requiring a profile file for input of the activation-time selections, or runtime menus, or other capabilities. As another example, "late" binding times (i.e., activation-time and runtime) might require more general and robust access to data, rather than less general, optimized data structures which would suffice for compile-time binding. As the technology of window managers progresses, features which now may be thought of as *compile-time* will migrate toward being runtime features. Some more advanced window managers such as gwm [Nahaboo 89] have already done this to some extent, making features that have always been assumed to be compile-time options (of necessity) into runtime choices.

Another option to pursue is that of widening the set of possible binding times to include "system design" phase binding as a subdivision of compile-time binding, where system design binding indicates more far-reaching architectural decisions. For example, in such an approach different values of compile-time features may all have source code available, but only one can exist in the executable. This area must be addressed more completely, with *feature binding time* being fully incorporated into the feature model.

8.2.2. Long-Term

8.2.2.1. Justifying Domain Analysis Economically

The intuitive justifications for performing domain analysis are the same as those which justify software reuse: improved quality and reduced cost. The specific questions which a system engineer contemplating the use of domain analysis must have answered are:

- What will the up-front costs be?
- What kind of return on investment can be expected, and over what period?

In order to properly answer these questions metrics must be collected on the effort expended in applying a domain analysis method such as FODA. This must be followed by the use of the domain analysis products on subsequent efforts, with a measurement of the impact they have. Until this has been done, the benefits of domain analysis are theoretical, and the work done must be treated as research.

8.2.2.2. Automating Support of the User Decision-Making Process

The domain analysis should provide information that supports the user's decision-making process as the user specifies and implements a new system. This information should include a standard set of products from which to build systems. (These products are called *domain products* in the FODA method.) To support decision-making, these products must include performance and cost assessments. The process of applying these products must be grounded in existing methods for engineering design [Cross 89], providing support both during:

- Divergence: when a wide range of alternatives are under evaluation.
- Convergence: when the final, evaluated specification is complete.

An approach from the design engineering discipline commonly used to explore established design alternatives and create a new design is the *morphological method*. In this approach, the design of a related class of systems is broken down into a set of sub-problems that are common to the entire class. For any new system in the class there are known means of solving the sub-problems. The design process becomes one of recognizing the commonality of a new system with other previously developed systems, and choosing the appropriate methods of implementing each of the sub-problems. Design reuse is further advanced because many of the same sub-problems generally apply to more than one class of systems. Tools to support this process may incorporate expert systems to support design constraints [Maher 86].

Domain analysis can support a similar process for software development. By establishing the set of sub-problems (the feature model) and solutions to these problems (the functional and architecture models), domain analysis supports both divergence and convergence. In addition, development of design rationales, with issues and arguments, addresses issues in design constraints. Merging this approach with already established design engineering approaches can be an effective means of supporting the decision making process.

8.2.2.3. Merging Domain Analysis and Other Reuse Methods

Domain analysis can also provide guidance in determining what to build to support reuse and how to build it. By agreeing on a common model for development, developers of reusable software can produce software to meet a range of needs for a set of problems. Instead of filling a repository with a large number of general purpose, non-integrated components, the pool of reusable resources will consist of tested and measured solutions to specific sub-problems in a given application area. In addition, the range of capabilities of systems in the area (i.e., the features) will determine customization requirements. Reusable software developers will know what to build and how to parameterize their products for varied use across the domain, rather than overgeneralizing them for all possible contexts.

The products of the FODA method also provide a natural organization for the library. The features/functional models define the structure for organizing and populating a software reuse library. The models allow users to see a recommended structure for solving their problem, while finding out about the software available to implement the solution. The customer for a new system is asked to define his software requirements in light of capabilities suggested by the domain products. The developer can then work with the customer to establish appropriate means for delivering those capabilities from existing software. Cost and performance factors that have been established for existing solutions can be compared, and the customer-designer interaction becomes one of negotiation in selecting the best choices that meet the customer's requirements.

9. Conclusions

Domain analysis is a necessary first step in establishing requirements for software reuse. The analysis can serve a variety of purposes toward this end:

- specifications for reusable resources
- tool support (e.g., catalogs, constructors)
- process model for reuse

In general, the analysis provides a detailed overview of the problems solved by software in a given domain.

Methods for domain analysis must take both the products and process into account. Those methods that have generated successful results have been tested in realistic domains. In some cases the analysis can lead to tools that can construct entire applications. Other methods are geared towards coverage of a number of domains, without providing detailed constructor tools.

The Feature-Oriented Domain Analysis (FODA) method is built on the results of other successful domain analysis endeavors. The method establishes three phases of a domain analysis:

- Context analysis: to establish scope
- Domain modelling: to define the problem space
- Architecture modelling: to characterize the solution space

The FODA method establishes a basis for proper domain scoping, which is critical to success. Without appropriate scoping the results can be too diffuse to meet the needs of application developers, or too narrow and omit critical areas of a domain. The feature model of the FODA method is central to domain modelling; features parameterize all other models. They provide a description of any real or proposed systems by means of sets of feature values. The domain model is refined by additional characteristics of the domain:

- Composition rules, also a part of the feature model, constrain combinations of features. They provide a means of developing automated support tools to validate complete and consistent sets of feature values.
- The functional model provides a *behavioral* and *functional* view of the system. Features are used to parameterize this model to support alternative views.
- The rationale for selecting options is supported by "Issues and Decisions." These are built around specific features or around aspects of the functional model.

Tool development/adaptation must take into account domain analysis methods. The complexity of even a well understood domain, such as that of window managers, establishes the need for tools to handle the volume and variety of information a domain analysis can generate.

The FODA method will continue to evolve through subsequent domain analyses. While the sample analysis covered the first two phases of the method, performing the architecture modelling is required to test the FODA method in that phase. The sample analysis pointed out the need for additional support in the area of handling the representation of commonality, both generalization and specialization. The FODA method must also be expanded to provide automated support for rationales to support user decisions. Applying the FODA method in new domains will support the evolution of the method and give further validation to the approach.

Appendix A: Forms

A.1. Feature Definition Form

```
Name: <standard feature name>

Synonyms:  <name> [FROM <source name>]
```

One or more synonyms may be defined, and the source of each name may optionally be included.

```
Description:
<textual description of the feature>

Consists Of <feature names> [ { optional | alternative } ]
```

This information shows the hierarchical structure of features, and may be represented graphically.

```
Source:
        <information source>
```

This information is used to produce a feature catalog. The source of information (e.g., standards, textbooks, existing systems) from which the feature is derived is included here.

```
Type: { compile-time | load-time | runtime }

[Mutually Exclusive With: <feature names>]
[Mandatory With: <feature names>]
```

A.2. Entity Description Form

```
Entity: <entity name>

Synonyms: <synonyms>

Description:
<a textual description of the entity>

Attributes:

<attribute name>:  <value range> [<unit>]

Source:
<information source>
```

The source of information (e.g., standards, textbooks, existing systems) from which the feature is derived is included here.

A.3. Attribute Description Form

```
Attribute: <attribute name>

Synonyms: <synonyms>

Description:
<a textual description of the attribute>

Value range:
<value range specification> [ <unit name>]

Source:
<information source>
```

The source of information (e.g., standards, textbooks, existing systems) from which the feature is derived is included here.

A value range can be any combination of:
(1) value types such as integer, string, real, boolean
(2) range of values (e.g., 10 through 100)
(3) strings (e.g., South, North)

Examples of the <unit name> are days, pounds, seconds, etc.

A.4. Relationship Type Forms

```
        ..................................................

Relationship Type <name>

Description:

        <A textual description of the semantics of the
        relationship type. Any rules applied to the
        relationship type must also be included.>


Parts: (<role name> { <entity type name> ... |
                      <attribute name>       |
                      ANY-ENTITY}            ;) ...
```

The Parts statement defines the roles of the entities in a relationship and what types of entities can play each role. For example, the Activity hierarchy relation of Statemate can be defined as:

```
                Parts: upper-part Activity
                       lower-part Activity;

Connectivity:
              (<role name>   {ONE | MANY} ) ... ;
```

The Connectivity defines how many entities can play the same role in a relationship. The example above, could be specified as:

```
                Connectivity:  upper-part ONE
                               lower-part MANY;
```

to indicate that one Activity may contain many Activities and that one Activity cannot be contained in other Activities more than once.

```
Syntax:
              (<role name> (<keyword> ... <role name> ... ;) ...
```

The Syntax statement is used to define a language statement for expressing the relationship in text. A graphical language may also be defined.

A.4.1. Relationship Type *is-a* Form

```
Relationship Type is-a ;

Description:
        To describe generalization/specialization relationships
        between entities. An entity in a generalization/
        specialization hierarchy inherits all of the attributes
        of its generalization entity. The value of an inherited
        attribute may be modified as long as the modified value
        is within the range of its generalization entity's value.
        A specialization entity may have attributes that are not
        defined for its generalization entity.;

Parts:  generalization  ANY
        specialization  ANY;

Connectivity:
        generalization  MANY
        specialization  MANY;

Syntax:
        specialization IS-A generalization;
        generalization IS-A-GENERALIZATION-OF specialization;
```

A.4.2. Relationship Type *consists-of* Form

```
Relationship Type consists-of;

Description:
To specify an aggregation relationship between an
entity and its constituent parts.;

Parts:  whole ANY
        parts ANY
        how-many INTEGER;

Connectivity:
        whole MANY
        parts MANY
        how-many ONE;

Syntax:
        whole CONSISTS OF [how-many] parts;
        parts IS A PART OF whole;
```

A.5. Issue Description Form

```
        ------------------------------------------

        Issue: <issue-name>

        Description:
        <a textual description of the issue>

        Raised at: <component name>
```

The "Raised at" statement indicates the component (e.g., an Activity of Statemate or a feature in the feature model) during the refinement of which the named issue was raised.

Decision: <decision name>

Description:
<a textual description of the decision>

Rationale:
<a textual description the rationale behind the decision>

Constraints/Requirements:
<a textual description of any new constraints caused by, or any new requirements derived from the decision>

Applies to: <component name>

The "Applies to" statement identifies the components that are resulted by the decision.

Appendix B: Domain Terminology Dictionary

In this dictionary terms which have been designated as "standard" for the purposes of the domain analysis appear in bold type, with a list of synonyms following. Entries for the synonyms point back to the standard entries.

above: See *expose*

abstract data object: A generic view of a widget, which allows various styles of user inputs to be transformed into simple data values

accelerator: (also called *shortcut*)
A way for an experienced user to bypass cumbersome novice commands to allow faster operation. A *keyboard equivalent* of a menu command is a type of *accelerator*.

active grab: (see also *grab* and *passive grab*)
A grab is *active* when the pointer or keyboard is actually owned by the single grabbing application

active window: See *input focus*

active window selection: See *input focus selection*

ancestor: (also called *superior*)
A window is an ancestor of any and all of its descendants.

application programmer interface (API):
The set of subroutines and calling conventions for the programmer's interface to the window system

application: An executable process

atom: A unique ID corresponding to a string name. Atoms are used to identify various pieces of information within the window manager.

auto exposure: See *auto raise*

auto raise: (also called *auto exposure*)
The ability of the window to go to the top of the current stacking order of overlapping windows as soon as it becomes the input focus, without an explicit expose operation.

back: See *hide*

backing store: (see also *save under*)
When the contents of a window (or a region of a window) which is *obscured* (by an overlapping or transient window) are saved for later restoration, the temporary area used to save the pixels is called a *backing store*.

background: The background color or tile pattern of a window upon which all text oor graphics are displayed

base bar: See *footer*

base window: The main window associated with an application process

below: See *hide*

bitblt: (pronounced *bit-blit*) An abbreviation of "*bit bl*ock *t*ransfer" which is an operation that

moves a rectangle of pixels from one area of a pixel-based screen to another.

bit gravity: (see also *window gravity*)
The way the contents of a window are attracted to a particular position within a window relative to a corner or edge during a resize operation.

bitmap: A (usually rectangular) portion of a pixel-based display where each pixel is represented by one bit.

border: (also called *frame*)
A *border* is the displayed edging around a window that can (depending on the window system) be used as both a command area and a visual feedback area. The border may be a solid color or a pattern, and usually has a variable width.

bottom: See *hide*

bounding box: (see also *ghost* and *select all*)
When multiple items need to be selected for an operation the *bounding box* paradigm may be used. A ghost box is drawn with, typically, the upper left-hand corner at the the point of origin and the lower right-hand corner rubber-banding to the pointer position. The items within the box are considered to be selected for the operation. This is most commonly used to mark portions of a graphic image for manipulation.

bury: See *hide*

busy feedback: See *dimmed*

button: See *mouse button*

callback: A type of procedure, usually bound to a widget, that is invoked by a user action, such as a click on a mouse button. Typically the window manager receives notification of the click via an event, looks for any procedures attached to the event, and uses a *callback* to have the application execute the attached procedure(s).

canvas: See *window*

caret: See *text cursor*

cascading menu: See *walking menu*

child window: (also called *subwindow*)
A window that has a parent window. A top-level window has the root window as its parent, and so all windows (except for the root window) are *child window*s, but some have different parents.

circulate: (also called *cycle windows* or *shuffle*)
Rotate through all displayed windows which overlap (in an overlapped window system), exposing the bottom window (or hiding the top window) and continue to cycle through all (overlapping) windows. Circulating windows by exposing the lowest is called *circulating up*, and the opposite is called *circulating down*.

click: Pressing and releasing a button (typically a mouse button) in rapid succession.

click-to-type: See *listener*

client: An application program that makes requests of the server to do things, such as draw windows, text, and other objects.

client server model: An architectural model that separates into two parts the job of drawing windows between the client and the server. The client and server execute asynchronously and communicate via a protocol. This model is used to improve device independence in some windowing systems.

client window: Strictly speaking, a window whose display is totally controlled by a client application. In more general terms, a subwindow controlled by any application, excluding any related subwindows such as the scroll bar, title bar, etc.

clipboard: A holding area where data is placed after it is "cut" from within a window system, and from which data may be "pasted".

clipping: Trimming the output of a window so that it only goes into the areas that need to be updated. For example, if a window is half-covered, when it is refreshed it only needs to redraw half of itself, and the other half is *clipped*.

clip mask: See *clipping region*

clipping region: (also called *clip mask*)
The area of a window that is visible to the user after the window's size, shape, and other overlapping windows are considered. The window manager clips the output from the window to this shape before displaying it on the screen. With rectangular shaped windows the clipping region may be defined with a bitmap or a list of rectangles.

close: See *iconify*

collapsed window: See *icon*

color cell: A 3-tuple of values specifying red, green, and blue intensities.

color map: A set of color cells which, put together, constitute all of the colors which are currently displayable.

command button: A widget that resembles a button and contains text or graphics to indicate the function of the button. When the mouse enters the window it is highlighted, and clicking the mouse while in the button notifies the application.

composite widget: A widget which is composed of other widgets

confirmer: See *confirmation box*

confirmation box: (also called *confirmer*)
A pop-up menu widget that offers a binary choice to the user, typically *yes* and *no*, or *cancel* and *continue*

containment: (see also *in*) A window *contains* the pointer if the hotspot of the pointer is within a visible region of the window or a visible region of one of its child windows.

context-sensitive borders: Button presses and releases on a window border will be interpreted as commands to the window

control: See *widget*

corner: (see also *handle*) A pointer-sensitive area of a window located at each of a window's corners. A *corner* may be used in move and resize operations.

covered presentation: See *overlapped presentation*

cover: See *hide*

current window: See *input focus*

cursor: A locator of information on the screen, presented as a small graphical image (a *cursor picture*) superimposed on the screen and controlled by a *pointing device*.

cursor picture: (also called *tracking symbol*)
The specific image used to show the location of the pointer, often to indicate the current function the pointer is performing. Different cursor pictures may be associated with different windows.

cut buffer: See *cut-and-paste buffer*

cut-and-paste: (see also *selection*)
To delete a portion of a text or graphics window display and later reinsert it into the same or another window

cut-and-paste buffer: (also called *cut buffer*)
A temporary buffer belonging to the window manager used to store the contents being transferred in curt-and-paste operations

cycle windows: See *circulate*

deiconify: (also called *expand, open,* or *restore*)
Change an icon into its corresponding main window

depth: The number of bits available for a pixel to represent its shade or color in a pixel-based imaging model.

descendant: (also called *inferior*)
The descendants of a window are all of the child windows nested below it: the children, the children's children, etc.

desktop metaphor: Either 1: an intuitive user interface metaphor used with overlapped window managers that makes an analogy between overlapped windows and papers laying stacked on a physical desk top, or 2: a window manager that uses icons to represent files and directories rather than executing processes, in addition to the overlapped appearance, so that there is an even stronger resemblance to a physical desktop. The MacIntosh is generally considered to use a *desktop metaphor*.

display: A set of one or more screens that are all driven by a single server

dialogue box: (also called *form*)
A window widget that may appear dynamically to inform the user of an event and possibly ask for input.

dimmed: (also called *busy feedback*)
When a widget or object is currently either busy or inactive it is *dimmed* (i.e., typically grayed or displayed in a gray font, rather than black) to show that it can't accept input.

dock: (see also *icon box*) A special area on the screen where icons are kept

double-click: (see also *click*)
Two clicks in rapid succession

drag: To press a button (typically a mouse button) and hold it down while moving the mouse. This is sometimes used to move an object on the screen (such as a window or icon) by visibly *dragging* it across the screen to its new position.

drawable: Windows and pixmaps are collectively known as *drawables*.

draw through: Selecting a region of text by moving the pointing device through it.

drop-down menu: A type of menu widget

drop shadow: An image that resembles a shadow falling from the window onto the screen background behind it. Its ostensible purpose is to enhance the outline of the window.

elevator scroll bar: See *scroll bar*

end style: The shape used to terminate the end of a line: rounded, square, etc.

event: An occurrence that typically causes action within a window system, such as input, output, an error, etc. Events are grouped into types, and are usually reported relative to a window.

event mask: The set of event types that is requested relative to a window is described by an *event mask*.

event propagation: Device-related events propagate from the event source window to ancestor windows until interest is expressed in handling that type of event or until the event is discarded explicitly.

event source: The smallest window *containing* the pointer is the source of a device-related event.

expand:

1. See *deiconify*
2. See *full screen*

expose: (also called *above, top, front,* or *raise*)
To change the stacking order of a window in an overlapped system so that no other window obscures any part of it. After a window has been *exposed*, it may then obscure other overlapping windows.

exposure event: An event caused by the exposure of a window. This event informs the application that contents of regions of its window have been lost by having been previously obscured.

fixed menu: A menu which always appears at a fixed place on the screen or within a window.

fill pattern: See *stipple*

focus of control indicator: This visual feedback device tells the user whether or not this window is the input focus. Examples of such indicators are highlighted borders, cursors, scroll bars, title bars, exposing the focus window, and shading all other windows.

follow-the-pointer: See *real estate*

font: A font is an array of glyphs along with information about the sizes of its glyphs.

footer: (also called *base bar*, see also *title bar*)
A region displayed immediately below a window that is used for information and error messages

foreground: The color or tile pattern that is used for drawing text or graphics; in text display the foreground is often black and the background is white.

form: See *dialogue box*

frame: See *border*

front: See *expose*

frozen events: Event processing may be *frozen* while the screen is being changed. As an example, if a button is pressed to indicate that a menu should appear, then events may be *frozen* to insure that the menu is drawn before the button-release event is processed. This means that the event is guaranteed to be reported with respect to the menu windows, and not those underneath where the menu now lies.

full screen: (also called *expand* or *maximize*) (see also *zoom*)
The state of a window that occupies the entire screen display, or the action that resizes and moves a window so that it reaches that state

gadget: A type of *widget* that is not tied to a window, but is rather an independently executing process

gauge: (also called *thermometer*)
A read-only display of a value that resembles a thermometer

geometry manager: A mechanism that can be used to define the layout of a window and its children, as well as determine the action to be taken when a resize of the window is performed

ghost: (also called *window outline* or *hair-lines*)
Typically a rubber-banded outline of a window displayed while the position or size of the window is being changed by the user

glyph: A type of small image- font characters are glyphs

grab: The input devices such as the keyboard, pointing device, and mouse buttons may be temporarily *grabbed* for exclusive use by an application

graphical user interface (GUI): (also called *look and feel*)
The appearance and behavior of the system. Both OSF/Motif and Open Look are GUIs. A GUI may be realized through a combination of toolkit widgets, usage conventions, and window manager policy.

graphics context (GC): The collection of information that influences drawing operations, including the foreground and background, the *clipping region,* the line width and style, the plane mask, the tile and stipple, the *end style* and the *join style.*

graphics package: See *imaging model*

gravity: (see also *bit gravity* and *window gravity*)
An attraction between objects on the screen that imposes a precise alignment between them in response to an approximate user movement

grid:

1. See *ghost*
2. An automatic way of neatly aligning a set of windows or icons for the user

grow: See *resize*

hair-lines: See *ghost*

handle: (see also *corner*) A pointer-sensitive area of a window located at each of a window's sides. A *handle* may be used in move and resize operations.

header: See *title bar*

hide: (also called *below, bury, bottom, back, cover, obscure, occlude,* or *lower*)
To change the stacking order of a window in an overlapped system so that it obscures no other window.

hidden region: (also called *shared region*)
A region on the screen in an overlapped system which is used by two windows simultaneously

highlight: A visual feedback mechanism used to indicate a special state of something on the screen. *Highlighting* may be achieved by blinking, by reverse video on monochrome systems, or by special colors on color systems.

hint: A piece of information provided to the window manager about the recommended placement or sizing of a window which may or may not be honored.

history: See *saved lines*

hotspot: The specific point within a pointer's cursor picture whose coordinates are returned with a cursor event.

icon: (also called *collapsed window* or *icon window*)

1. A small graphical image that represents a window running an application. An icon of the window is used when the window is not currently in use, but will be used again. Use of the icon rather than the main window when it is inactive conserves screen real estate.
2. Any small image that is used to represent a concept to the user. An icon in this sense may be used to label a button widget instead of a text name.

icon box: (see also *icon manager*)
A window in which all icons are stored

icon manager: (see also *icon box*)
An alternative to normal separate icons for handling temporarily unused processes. *Icon managers* will list the names of the windows they are currently handling. Multiple icon managers may coexist simultaneously to handle different sets of windows.

icon window: See *icon*

iconic menu: A menu whose options consist of symbolic images representing the different functions which can be performed.

iconify: (also called *shrink, close, minimize,* or *iconize*)
Change a main window into its corresponding icon.

iconize: See *iconify*

illegal item: A menu item which is not valid in a certain context. In some window systems theses items are *dimmed* to indicate they cannot be chosen.

image: A graphical representation normally realized as a *pixmap*

imaging model: (also called *graphics package*)
The model used for displaying images, such as the pixel model.

in: The pointer is *in* a window if the window contains the pointer but no inferior contains the pointer.

inferior: See *descendant.* May also refer to a window's position in the *stacking order* as not being the top window.

input device: Allows the user to provide input to the system in various forms such as positions, text, and values

input focus: (also called *active window* or *current window*)
The window that is the current recipient of user input. See *keyboard focus*.

input focus selection: (also called *active window selection*)
The way the window manager allows the user to choose a window as the current input focus

input manager: Accepts input events and transmits them to the process manager

inputonly window: A window which cannot display any output- *inputonly* windows are invisible, but can handle input differently from other windows. Typically a different cursor picture is used for the pointer so that the user understands that input will be handled differently. *Inputonly* windows cannot have *inputoutput* windows as children.

inputoutput window: A normal main window (one which is not an *inputonly* window)

insert point: The point in a window where new keyboard input will be inserted. This point is designated by the *text cursor*.

intrinsics: The foundation layer of functionality used to manipulate widgets

join style: The shape used to connect two lines at a corner: rounded, flat, closed, etc.

joystick: A type of pointing device

kernel: The low-level interface to the underlying graphics hardware and operating system

key grabbing: Keys on the keyboard may be passively grabbed. When the key is pressed, the keyboard is then actively grabbed.

keyboard: A keyboard is a set of keys that allows the entry of textual data.

keyboard equivalent: (see also *accelerator*)
A single key or a sequence of keys that performs an operation which is also accessible from a menu

keyboard focus: (see also *input focus*)
The window that is the current recipient of all keyboard user input.

keyboard grabbing: The keyboard may be actively grabbed by an application, and then key event will be sent to that application, rather than to the usual application indicated by the input focus.

knob: A type of physical input device that functions as a valuator

label: The displayed name or title of a menu item, button, or other widget

light pen: A type of pointing device

lightweight process: (also called *thread*)
A type of process that lacks the normal full process context, which allows fast context switching

list menu: A menu widget which appears on the screen as a sequence of lines, each of which contains an item; it can be of the *fixed* or *pop-up* variety

listener: (also called *click-to-type*)
A listener style window manager sets the input focus to a particular window when that window is clicked on with a mouse button.

locked menu: A menu where the user is not allowed to change the current item selection

look and feel: See *graphical user interface*

lower: See *hide*

main window: A window displayed in its normal, full form-- i.e., *not* an icon

menu: A widget that is a finite list of text buttons called *menu items*; choosing an item from a menu typically invokes some action

mapped: (see also *unmapped* and *viewable*)
A window is *mapped* if the window manager has designated that it should be *viewable* to a user. For example, while a window is being used, its corresponding icon is *unmapped*, and therefore not *viewable*.

maximize: See *full screen*

menu item: One of the possible choices on a menu

minimize: See *iconify*

modifier key: A key which, when pressed in conjunction with or preceding another key, does not send *another* character, but instead alters the character that is sent.

monitor: A type of physical output device

mouse: (see also *pointer*) A pointing device that is held in the hand and moved across a surface.

mouse-ahead: Giving commands with the pointer before the system is ready to execute them; these are queued for later execution

mouse button: (also called *button*)
A physical input device with two states, *up* and *down* (or *released* and *pressed*)

mouse cursor: See *pointer*

multiclicking: Clicking a button two or more times in rapid succession.

multitasking: The ability of an operating system to simultaneously execute more than one process. The UNIX operating system is multitasking; the DOS operating system is not.

object-action model: (also called *select-then-operate paradigm* or *subject-verb model*)
A user interface model where the user must first select an object to operate on (a window in a window system) and then an action to perform on it

output device: Converts data into a human-understandable format such as displayed text and graphics, or sound

obscure: See *hide*

occlude: See *hide*

opaque: Refers to the way a window is represented during a move or resize operation. An *opaque* window means that the full window image is preserved and continually displayed during the operation so that the full effect of the operation may be seen by the user. This is typically supported only on relatively fast display hardware.

open: See *deiconify*

overlapped presentation: (also called *covered presentation*)
A window system paradigm where a window may occupy any position on the screen, and may overlap or be overlapped by any number of other windows

pages: See *stacked menu*

paint and stencil model: An imaging model used in some window systems (such as NeWS). *PostScript* is one implementation of a *paint and stencil* model.

pan: (see also *scroll* when applied to text)
The ability to use a window as a viewport on a larger virtual graphic space, and move the viewport to focus on different parts of the space.

pane: A defined area within a window, typically the central or primary display of the window; multiple panes within a window are normally *tiled* and do not overlap one another.

parent window: A window that has child windows. The area of a child window can never extend beyond the borders of its parent.

passive grab: (see also *grab* and *active grab*)
Grabbing a key or mouse button is a *passive grab*. The grab activates when the key or mouse button is actually pressed.

pixel: A short form for "*pic*ture *el*ement"- a single point on a bitmap monitor which may be either colored or black and white.

pixel map: See *pixmap*

pixmap: (abbreviation of *pixel map*)
An array of pixels; alternately, a stack of bitmaps, or a three-dimensional array of bits.

pixel model: (also called *raster model*)

An imaging model based on pixels and logical operations upon pixels, such as *AND*ing and *OR*ing pixels together. Within a system based on the *pixel model* all images are generated using pixels.

point: Move the pointer to a specific region or object

point-to-type: See *real estate*

pointer: (also called *mouse cursor*)
A type of cursor that marks the current position of the pointing device.

pointing device: (also called *mouse*)
An input device that allows the entry of accurate location data on the screen by moving the pointer on the screen

pop-up window: (see also *transient window*)
A window that appears dynamically to allow the user to enter some kind of input, and then disappears again after the input is made. A *pop-up* window is a type of transient window.

presentation: The representation of a window on the screen, which is either in iconic or main form.

process manager: Ties applications to windows for I/O and other operations

prompter: A pop-up window widget that requires the user to type in a response

property: Windows may have associated *properties* such as name, type, data format, and various data. Such information as resize hints, program names, and icon formats may be expressed as properties.

property list: The list of all properties defined for a window

pull-down menu: (see also *pop-up menu*)
A type of *pop-up menu* that appears after the selection of an item on the screen, usually immediately below, and allows the user to specify further options

pushpin: A *pushpin* is used to fix a transient menu into a position.

raise: See *expose*

raster model: See *pixel model*

real estate: (also called *point-to-type* or *follow-the-pointer*)
A window system paradigm where the input focus is always at the window the pointer is in

recursive subwindows: (see also *child window*)
Windows which adhere to a parent-child window hierarchy

redirecting control: A way of enforcing window layout policy- when an application attempts to change the size or position of a window, the operation may be redirected to another application

redisplay: See *refresh*

redraw: See *refresh*

refresh: (also called *redisplay* or *redraw*)
To erase and freshly display all or part of the screen that may have been damaged

region: The area of a polygon defined by a collection of points. Regions are usually used within windows to further subdivide the containing space for the application.

reparent: To *reparent* a window means to insert a new parent window behind an existing window, often for highlighting or additional user input.

reserved area: See *special area*

resize: (also called *shape* or *grow*)
To change the shape and/or the size of a window

resource: A *resource* is an object handled by the window manager, such as a window, cursor, font, color map, etc.

restore: See *deiconify*

root window: The window that has no parent window, and is the ancestor of all other windows on the screen. The root window covers the entire screen.

rubber band: The act of interactively moving the pointer in relation to a stationary point while lines are continuously drawn between them to show the possible final state of the relationship. *Rubber-banding* is often used to allow the user to visualize the results of a resize operation on a window.

save-set: A list of windows that will be restored to their normal state if the window manager exits abnormally (i.e., the windows will be mapped again if the window manager had unmapped them for an icon)

save under: (see also *backing store*)
The pixmap of the contents of a window (or a region of a window) which is *obscured* (in an overlapped window system)

saved lines: (also called *history*)
The lines of text in a window that have been scrolled off the window, but have been saved and are still available for display.

screen: A logical output display device on which windows and other graphics may be displayed

screen real estate: The visible area provided by a screen. A window manager's function is to manage this area.

screen manager: Directs output to the proper window on the screen.

scroll: To move through a set of data which is larger than the available window by alternately showing different portions of it.

scroll bar: (also called *elevator scroll bar*)
A composite widget usually associated with a window that allows the user to specify the scroll amount using valuators and command button widgets.

select all: (see also *bounding box*)
When multiple items need to be selected for an operation the *select all* paradigm may be used to allow the user to individually select each item.

select-then-operate paradigm:
See *object-action model*

selection: A *selection* is an item of data which can be highlighted in one instance

of an application and pasted into another instance of the same or a different application.

server: A program that runs on each user's workstation and draws the required objects on the display in a client-server architecture

shadow: (see also *drop shadow*)
A shaded area on the screen sometimes used to show the previous position of an icon that has been deiconified to its main window.

shape: See *resize*

shared region: See *hidden region*

shortcut: See *accelerator*

shrink: See *iconify*

shuffle: See *circulate*

sibling: Children windows of the same parent are known as *siblings*.

slide-off menu: See *walking menu*

slidebox: A rectangular area within a valuator widget that may be positioned with the pointer

slider: An input/output widget used to both set and display a value; *sliders* may resemble gauges.

special area: (also called *reserved area*)
A reserved region on the screen that is provided exclusively for special functions, such as the display of error messages, prompts, icons, etc.

stacked menu: (also called *pages*)
A type of *submenu* that stacks menus on top of one another but slightly offset so that the total effect is that of pages in a book. Each page will typically *auto raise* when touched by the cursor to display its options.

stacking order: The order of overlapping sibling windows (in an overlapped system) that determines which one lies visually on top and which on the bottom. Circulating, exposing, and hiding windows changes the stacking order. The first window in the *stacking order* is the window on top.

stipple: (also called *fill pattern*)
A bitmap that is used to *tile* a region (not related to *tiled presentation*)

stippling: Tiling a region with a stipple

subject-verb model: See *object-action model*

submenu: (see also *walking menu* and *stacked menu*)
A menu widget that displays an additional menu after an item in a previous menu is selected, providing additional options and parameters to that selected operation.

subwindow: See *child window*

superior: See *ancestor.* May also refer to a window's position in the *stacking order* as being the top window.

switch: A type of input device

tear-off menu: A pop-up menu which is fixed in a certain position with a *push-pin* and remains there.

temporary window: See *transient window*

terminal emulator: A window which emulates the functionality of a standard terminal such as a VT100.

text button: A widget that consists of a displayed piece of text which becomes highlighted when the pointer crosses over it. If a mouse button is pressed while the cursor is within the text button the application is notified.

text cursor: (also called *caret*)
A cursor that marks the current position for text insertion within a window. This cursor position is independent of the position of the *pointer*.

thermometer: See *gauge*

thread: See *lightweight process*

tile: A pixel map that can be repeated both vertically and horizontally to completely cover (or *tile*) an area.

tile mode: The position of a background tile pattern in a window can be placed relative to the parent window, or absolutely in the child without consideration of matching the parent's pattern.

tiled presentation: A window system paradigm where each window occupies a separate portion of the screen, and no window can overlap any other.

title bar: (also called *header*)
A region displayed immediately above a window that indicates the name or title of the window

titlebutton: A type of command button that is incorporated into the window title bar and is typically used for moving or resizing windows

toolkit: A set of facilities for building user interfaces for applications, typically including widgets for menus, forms, text editing, and scrolling capabilities

top: See *expose*

top-level window: A window that is a direct child of the root window

touch panel: An input device that uses the touch of the user's finger or a stylus on the screen to mark a position

touch tablet: A pointing device that uses the touch of the user's finger or a stylus on a tablet to mark a position or move the pointer

track ball: An input device that remains stationary and is used to move the pointer

tracking symbol: See *cursor picture*

transient window: (also called *temporary window*)
A temporary subwindow of the root window that is usually (in actuality) associated with a top-level window, rather than the root. Having the window be a temporary subwindow of the root means that it will not be clipped by the borders of the top-level window which is its *real* parent. Transient windows typically use *save-unders* to prevent redrawing any underlying windows.

typescript package: Provides basic text editing capability for typing in command input, such as backspace, delete, etc.

unmapped: (see also *mapped*)
A window is *unmapped* when it is not *mapped*. An *unmapped* window is not *viewable*.

user interface language (UIL):
A language used to determine the user interface of an application by defining the functional characteristics

user interface management system (UIMS):
A system that provides facilities for the high-level design of user interfaces. Provides an additional layer of abstraction above a window system toolkit.

valuator: A widget sometimes used to implement scrolling bars that contains a rectangular *slidebox* that follows the motion of the mouse. The *valuator* informs the application whenever the user changes the position of the slidebox.

view: See *window*

viewable: (see also *mapped* and *unmapped*)
A window is *viewable* if it and all of its ancestors are mapped. The fact that a window is *viewable* does not imply that it is necessarily *visible*, since the window may be totally *obscured* by other windows.

visible: A region of a window is *visible* if someone looking at the screen can see it (i.e., it is not obscured by another window).

visual: A *visual* is a data structure defining the physical characteristics supported by a particular screen. In particular it defines the various colormap abstractions supported and the depth(s) of the display.

warp: To automatically place the pointer into a window immediately after it has been deiconified

walking menu: (also called *cascading menu* or *slide-off menu*)
A type of *submenu* where as the user slides the cursor off the right side of a menu an additional menu appears containing sub-options of the chosen command. All of the menus will disappear after the final options have been chosen.

widget: (also called *control*)
An object found in a UIMS toolkit that is used to construct user interfaces. *Widgets* include such graphical devices for input and output as *command buttons*, *valuators*, *gauges,* etc. A widget may exist as an object in a single-inheritance class hierarchy of other types of widgets.

window: (also called *view, wob,* or *canvas*)
A logical area on a bit-mapped display that is connected to at least one application process.

window configuration: The collection of information about the appearance of a window on the screen which consists of the size, position, border width, and stacking order.

window geometry: A window's position, shape, and size

window gravity: (see also *bit gravity*)
The way the children windows are attracted to a particular position

within a parent window relative to a corner or edge during a resize operation. Styles of gravity may include *static, center,* and the *compass points*.

window layout policy: A policy on how top-level windows will be placed on the screen

window management system:
Manages a set of display windows running asynchronous processes.

window manager: Performs operations on windows (*create*, *move*, *resize*, etc.) so as to allow the user to use the screen space effectively.

window outline: See *ghost*

wob (window object): See *window*

zap lines: Ghost lines which temporarily "flash" to follow a window or icon from its original position to its new position during move, resize, or (de)iconify operations

zoom:

1. See *full screen*
2. Display successively smaller (or larger) outlines of a window to suggest shrinking (or expanding) during Iconify (or Deiconify) operations

zoomed window: A window whose pixmap is displayed at a larger than normal magnification.

Appendix C: System Feature Catalogue

In the feature catalogue tables the following conventions are used:

1. `<Name>`: This is the name used for this feature on this system.
2. "`yes/no`": This feature does/does not exist on this system.
3. `<blank>`: No information was collected for this feature.
4. "`*`": Selection of this feature is a load-time or run-time option.
5. "`--`": This feature is inapplicable to this system (a composition rule with another feature excludes it).

System/Feature	X10/uwm	VMS Windows	SunView	Mac Windows
multObjSelect	no	no	no	yes
inputFocusSelect	*	listener	realEstate	listener
mandatoryFocus	no	no	no	yes
revertToNewFocus	--	nextOnStack	--	nextOnStack
selectOrder	actionObject	objectAction	objectAction	objectAction
windowLayout	overlapped	overlapped	overlapped	overlapped
partiallyOff	yes	yes	no	yes
tiledColumns	--	--	--	--
windowShape	rectangular	rectangular	rectangular	rectangular
hasIcons	yes	yes	yes	yes
specialAreas	no	no	no	yes
titleBars	*	yes	yes	yes
highlightedAreas	bord./title/cur.[1]	cursor	bord./cur.[1]	title/scroll[1]
changeFocus	yes	yes[7]	yes	yes[7]
createWindow	yes[3,7]	yes[3,7,8]	yes[3,7]	yes[3]
destroyWindow	yes[4]	Delete	Quit	yes[4]
exposeWindow	Raise	Pop to Top	Front	*Select*
hideWindow	Lower	Push Behind	Back	no
circulateUp	Circulate	no	no	no
circulateDown	no	no	no	Cycle Windows
moveWindow	Move[7]	yes[5,7,8]	Move	yes[5]
constrainedMove	no	no	yes	no
resizeWindow	Resize[7]	Change Size[7]	Resize	yes
corners&Handles	all	all	all	lower right
expandWindow	no	no	Full Screen	yes
iconify	(De)iconify[7]	Shrink[7,8]	Close[7]	yes
deiconify	(De)iconify[7]	yes[7,8]	Open[7]	yes
refreshWindow	yes	no	Redisplay	no
refreshAll	Refresh	no	Redisplay	no
abortMove	no	no	*Cancel*	no
quitWindowSystem	no	no	Exit	no

Table C-1: Window Manager Features I

System/Feature	**Andrew**	**Symbolics**	**X11/uwm**	**OSF/Motif**
multObjSelect	no	no		
inputFocusSelect	realEstate	listener		*
mandatoryFocus				no
revertToNewFocus				--
selectOrder	objectAction	actionObject		objectAction
windowLayout	tiled	overlapped	overlapped	overlapped
partiallyOff	no		yes	yes
tiledColumns	yes	--	--	--
windowShape	rectangular	rectangular	rectangular	rectangular
hasIcons	yes	no	yes	yes
specialAreas	no	yes	no	no
titleBars	yes		*	yes
highlightedAreas	title[1]	greyed[2]	bord./cur.[1]	bord./title/cur.
changeFocus	yes	yes[7]	yes[7]	yes
createWindow	yes[3]	yes[3]	yes[6]	yes
destroyWindow	Quit	Kill	yes	Quit
exposeWindow	--	*Select*[7]	Raise	no
hideWindow	--	Bury	Lower	Lower
circulateUp	--	no	Circulate Up	Shuffle Up
circulateDown	--	no	Circulate Dn.	Shuffle Down
moveWindow	yes[5]	Move	Move	Move
constrainedMove				
resizeWindow	Enlarge	Shape	Resize	Size
corners&Handles	no		all	all
expandWindow	*Proportion*[6]	Expand		Maximize
iconify	yes	--	(De)iconify	Minimize
deiconify	yes	--	(De)iconify	Restore
refreshWindow	Redisplay	Refresh	no	no
refreshAll			no	yes
abortMove				no
quitWindowSystem	Logout			Quit

Table C-2: Window Manager Features II

NOTES

1. *Bord.* refers to a highlighting (or shading) of the border around the window, *title* refers to a highlighting of the window's title bar, *scroll* refers to the existence of the graphics in the scrollbar, and *cur.* refers to a highlighting of the text cursor in the window.
2. *Greyed* signifies that all portions of the screen other than the input focus window are overlaid with a grey stipple.
3. Few window managers have explicit *create* commands- a window is implicitly created when its primary associated application is started.
4. Few window managers have explicit *destroy* commands- a window is destroyed when its primary associated application is terminated.
5. These systems do not have explicitly named *move* functions- moving a window is accomplished by clicking on the border or title bar and dragging the resulting ghost.
6. The *proportion* command proportions all of the windows in a column so that they share the available space equally. It is analogous to *expand* since it makes a window as large as it can be without sacrificing the other windows.
7. These commands move the window to the top of the stacking order.
8. These commands change the current input focus.

Appendix D: Issues and Decisions

Issue: Input Focus Selection

Description: The way the window manager allows the user to designate the current input focus

Raised at: `inputFocusSelection`

Decision: `realEstateMode`

Description: A window becomes active by moving the cursor over it (if it is visible).

Rationale: Simple and quick to use

Decision: `listenerMode`

Description: The user must click on a window to make it active.

Rationale: Helps prevent accidental activation of the window. Allows the system to make the existence of a current input focus mandatory.

Issue: Window Layout

Description: The way windows relate physically on the screen.

Raised at: `windowLayout`

Decision: `tiledLayout`

Description: A window system paradigm where each window occupies a separate portion of the screen, and no window can overlap any other.

Rationale: Avoids dealing with issues of exposing and hiding overlapping windows, and makes it simpler to identify the recipient of user input (complex clipping masks are avoided). However, a tiled system must provide some sort of automatic layout system for the windows. In this paradigm, *resizing* events become common.

Decision: `overlappedLayout`

Description: A window system paradigm where a window may occupy any position on the screen, and may overlap or be overlapped by any number of other windows

Rationale: Provides a user interface that emulates the overlapping papers found in a physical desktop environment. The user is in almost full control of the window layout, but an overlapped system must deal with maintaining the stacking order of the window and the clipping masks for displaying their contents. In this paradigm, *exposure* events become common.

Issue: Tiling Method

Description: The type of window tiling which is enforced

Raised at: `tiledLayout`

Decision: `tiledColumns`

Description: Allow windows to be positioned only within a set number of columns.

Rationale: Simpler than the alternative, and it avoids potentially unpredictable screen layout changes when windows are added or removed.

Decision: `tiledArbitrary`

Description: Allow windows to be positioned anywhere on the screen (as long as no other window is overlapped).

Rationale: Intuitively less constraining than using columns to avoid overlap

Issue: Window Shape

Description: The set of shapes allowed for windows by the window manager

Raised at: `windowShape`

Decision: `rectangularShape`

Description: A window may only have a rectangular shape

Rationale: Relatively simple to compute which window the pointer is in, and the clipping regions for display

Decision: `arbitraryShape`

Description: A window may have any arbitrary two-dimensional shape

Rationale: If the window information is not rectangular (i.e., a clock face) then there is no wasted space from fitting a circle into a square.

Issue: Interactive Feedback

Description: The way the window manager shows the user the current size or shape of a window being moved or resized.

Raised at: `interactiveFeedback`

Decision: `ghostFeedback`

Description: An outline and/or grid of the window is drawn and moved with the cursor to the new location, where the complete window is drawn (and the old window erased.

Rationale: Provides sufficient user feedback for positioning and resizing, and requires significantly fewer resources than redrawing the entire image often enough to follow the moving cursor.

Decision: `opaqueFeedback`

Description: The window manager moves or resizes the entire original image of the window.

Rationale: Opaque moving and resizing allows the user to see immediately what the window will look like in the new position or shape, and is typically used on fast displays where the act of updating a potentially complex window display is feasible.

Issue: Command Selection Order

Description: The order in which objects and actions are selected for the performance of window operations.

Raised at: `selectionOrder`

Decision: `objectAction`

Description: The window is selected before the command.

Rationale: Selecting the object first allows the menu of possible actions to be tailored based on the type of object, i.e., an icon or a window. This makes the user interface more intuitive.

Decision: `actionObject`

Description: The command is selected before the window.

Rationale: Allows the possible actions to include commands that do not require the selection of a destination object.

Appendix E: Window Manager Features

`abortMoveOp`: optional
Abort the current move operation.
Parent: `moveWindowOp`
Source: *SunView window system experience*

`abortResizeOp`: optional
Abort the current resize operation.
Parent: `resizeWindowOp`
Source: *SunView window system experience*

`actionObject`: alternative
First the command, then the window.
Parent: `selectionOrder`
Source: *Open Look GUI documentation*

`activeIcons`: alternative
An icon may update its image.
Parent: `iconIO`
Source: *A Taxonomy of Window Manager User Interfaces*

`arbitraryNumberColumns`: alternative
Window layout is tiled using arbitrarily many columns.
Parent: `tiledColumns`
Source: *A Taxonomy of Window Manager User Interfaces*

`arbitraryShape`: alternative
Windows may be of any shape.
Parent: `windowShape`
Source: *NeWS window system documentation*

`borderHighlight`: optional
Highlight the border.
Parent: `highlightedAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`changeFocusOp`: mandatory
Change the input (keyboard) focus.
Parent: `windowManager`
Source: *General*

`circulateDownWindowsOp`: optional
Circulate down through all windows.
Parent: `stackingOrder`
Source: *X11 Xlib documentation*

`circulateUpWindowsOp`: optional
Circulate up through all windows.
Parent: `stackingOrder`

Source: *X11 Xlib documentation*

`colorHighlight:` alternative
Highlighting via a color.
Parent: `highlightMethod`
Source: *Motif window manager experience*

`commandAreaHighlight:` optional
Highlight the command areas.
Parent: `highlightedAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`constrainedMove:` optional
Window movement may be constrained horizontally or vertically.
Parent: `moveWindowOp`
Source: *SunView window system experience*

`cornersAndHandlesResize:` alternative
Users may select the windows corners and handles to resize it.
Parent: `resizeInput`
Source: *A Taxonomy of Window Manager User Interfaces*

`createIconified:` optional
Create new windows as icons.
Parent: `createWindowOp`
Source: *X11/twm documentation*
Rules: requires `hasIcons`

`createWindowOp:` mandatory
Create a new window.
Parent: `windowManager`
Source: *General*

`dataIcons:` alternative
Icons represent data items such as files and directories.
Parent: `iconUsage`
Source: *A Taxonomy of Window Manager User Interfaces*

`deiconifiedIconDisplay:` mandatory
The way currently deiconified icons displayed.
Parent: `hasIcons`
Source: *X10/uwm window manager experience*

`deiconifyIconOp:` mandatory
Turn an icon into a main window.
Parent: `hasIcons`
Source: *General*

`destroyWindowOp:` mandatory
Destroy an existing window.
Parent: `windowManager`

Source: *General*

`dimmedDeiconifiedIcons`: alternative
They are dimmed.
Parent: `deiconifiedIconDisplay`
Source: *Macintosh window system experience*

`dimmedHighlight`: optional
Dim all other windows.
Parent: `highlightedAreas`
Source: *Symbolics window system experience*

`dropShadowsEffect`: optional
Aesthetically pleasing 3D visual effect to make windows stand out.
Parent: `windowManager`
Source: *X11 Xlib documentation*

`eraseAfter`: alternative
Erase at the end of the move.
Parent: `moveErasure`
Source: *VMS window system experience*

`eraseBefore`: alternative
Erase at the start of the move.
Parent: `moveErasure`
Source: *VMS window system experience*

`expandWindowOp`: optional
Expand a window to fill the entire screen.
Parent: `resizeInput`
Source: *SunView window system experience*

`exposeAfterChangeFocus`: optional
Expose the window at the end of a change focus operation.
Parent: `changeFocusOp`
Source: *SunView window system experience*
Rules: requires `overlappedLayout`

`exposeAfterDeiconify`: optional
Expose the window at the end of a deiconify operation.
Parent: `deiconifyIconOp`
Source: *SunView window system experience*
Rules: requires `overlappedLayout`

`exposeAfterIconify`: optional
Expose the icon at the end of an iconify operation.
Parent: `iconifyWindowOp`
Source: *SunView window system experience*
Rules: requires `overlappedLayout`

`exposeAfterMove`: optional

Expose the window at the end of a move operation.
Parent: `moveWindowOp`
Source: *SunView window system experience*
Rules: requires `overlappedLayout`

`exposeAfterResize`: optional
Expose the window at the end of a resize operation.
Parent: `resizeWindowOp`
Source: *SunView window system experience*
Rules: requires `overlappedLayout`

`exposeWindowOp`: mandatory
Expose a window.
Parent: `stackingOrder`
Source: *X10/uwm window manager experience*

`fixedNumberColumns`: alternative
Window layout is tiled in a fixed number of columns.
Parent: `tiledColumns`
Source: *A Taxonomy of Window Manager User Interfaces*

`focusBeforeCommand`: optional
The need to make a window the focus before a command is performed on it.
Parent: `changeFocusOp`
Source: *Macintosh window system experience*

`ghostFeedback`: alternative
An outline displayed for feedback.
Parent: `interactiveFeedback`
Source: *X10/uwm window manager experience*
Rules: requires `moveErasure`

`hasIcons`: optional
The window manager supports icons.
Parent: `windowManager`
Source: *Symbolics window system experience*

`hiddenInputFocus`: optional
The ability to make a hidden window the input focus.
Parent: `updateHiddenWindows`
Source: *A Taxonomy of Window Manager User Interfaces*

`hideWindowOp`: optional
Hide a window beneath all other windows.
Parent: `stackingOrder`
Source: *X10/uwm window manager experience*

`highlightInputFocus`: optional
Highlight the current input focus.
Parent: `changeFocusOp`
Source: *A Taxonomy of Window Manager User Interfaces*

highlightMethod: mandatory
The way highlighting is done.
Parent: highlightInputFocus
Source: *Motif window manager experience*

highlightedAreas: mandatory
Where the current input focus is highlighted.
Parent: highlightInputFocus
Source: *A Taxonomy of Window Manager User Interfaces*

iconBox: optional
Icons are stored in an icon box (or manager).
Parent: hasIcons
Source: *Motif window manager experience*
Rules: requires iconBoxDeiconifiedIcons

iconBoxDeiconifiedIcons: alternative
Deiconified icons go into an icon box.
Parent: deiconifiedIconDisplay
Source: *Motif window manager experience*
Rules: requires iconBox

iconFocus: optional
An icon may be the current input focus.
Parent: activeIcons
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires changeFocusOp

iconIO: mandatory
The way icons deal with input/output.
Parent: hasIcons
Source: *A Taxonomy of Window Manager User Interfaces*

iconOperationFeedback: optional
Visible feedback for iconify/deiconify icon operations.
Parent: operationFeedback
Source: *X10/uwm window manager experience*

iconUsage: mandatory
The way icons are used in the window manager.
Parent: hasIcons
Source: *A Taxonomy of Window Manager User Interfaces*

iconifyWindowOp: mandatory
Turn a main window into an icon.
Parent: hasIcons
Source: *General*

inputFocusSelection: mandatory
The way the user selects the input focus.
Parent: windowManager

Source: *A Taxonomy of Window Manager User Interfaces*

`interactiveFeedback`: mandatory
The user feedback during a move or resize.
Parent: `moveResizeFeedback`
Source: *X10/uwm window manager experience*

`interiorHighlight`: optional
Highlight the interior of the window.
Parent: `highlightedAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`keyboardAccelerators`: optional
Keyboard equivalents for commands normally run from menus.
Parent: `windowManager`
Source: *A Taxonomy of Window Manager User Interfaces*

`largerThanScreenWindows`: optional
The ability to create windows larger than the screen.
Parent: `partiallyOffScreenWindows`
Source: *X10/uwm window manager experience*

`listenerMode`: alternative
Click to type.
Parent: `inputFocusSelection`
Source: *A Taxonomy of Window Manager User Interfaces*

`mandatoryFocus`: optional
The need for a designated input focus at all times.
Parent: `changeFocusOp`
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires `listenerMode`

`moveErasure`: optional
Erase the old window at start or end of move.
Parent: `moveWindowOp`
Source: *VMS window system experience*

`moveIcon`: optional
Move an icon to a new location.
Parent: `moveWindowOp`
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires `hasIcons`

`moveResizeFeedback`: mandatory
Visible feedback for move/resize operations.
Parent: `operationFeedback`
Source: *X10/uwm window manager experience*

`moveWindowColumnOp`: optional
Move a window from one column to another.

Parent: moveWindowOp
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires tiledColumns

moveWindowOp: mandatory
Move a window to a new location.
Parent: windowManager
Source: *General*

multipleInputFoci: optional
The ability to have multiple simultaneous input foci.
Parent: changeFocusOp
Source: *A Taxonomy of Window Manager User Interfaces*

multipleObjectSelection: optional
The ability to select multiple windows (or other objects).
Parent: windowManager
Source: *A Taxonomy of Window Manager User Interfaces*

newIconPlacement: mandatory
Where newly created icons are placed.
Parent: hasIcons
Source: *VMS window system experience*

nextOnStackingOrder: alternative
The input focus switches to the next on the stacking order.
Parent: revertToNewFocus
Source: *VMS window system experience*
Rules: requires overlappedLayout

objectAction: alternative
First the window, then the command.
Parent: selectionOrder
Source: *Open Look GUI documentation*

onePlaceResize: alternative
Users may select only the lower right corner to resize the window.
Parent: resizeInput
Source: *A Taxonomy of Window Manager User Interfaces*

opaqueFeedback: alternative
The complete window displayed for feedback.
Parent: interactiveFeedback
Source: *X10/uwm window manager experience*
Rules: mutex with moveErasure

openAreaIconPlacement: alternative
In an arbitrary open area.
Parent: newIconPlacement
Source: *VMS window system experience*

`operationFeedback`: mandatory
Visible feedback for users when operations are performed.
Parent: `windowManager`
Source: *X10/uwm window manager experience*

`overlappedLayout`: alternative
Overlapped presentation.
Parent: `windowLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`parentNewFocus`: alternative
The input focus switches to the parent.
Parent: `revertToNewFocus`
Source: *X11 Xlib documentation*

`partiallyOffScreenWindows`: optional
The ability to place windows partially off the screen.
Parent: `overlappedLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`passiveIcons`: alternative
*An icon may *not* display data from an application.*
Parent: `iconIO`
Source: *A Taxonomy of Window Manager User Interfaces*

`patternHighlight`: alternative
Highlighting via a pattern.
Parent: `highlightMethod`
Source: *X10/uwm window manager experience*

`pointerPositionIconPlacement`: alternative
The current pointer position.
Parent: `newIconPlacement`
Source: *X10/uwm window manager experience*

`processIcons`: alternative
Icons represent executing processes/applications.
Parent: `iconUsage`
Source: *A Taxonomy of Window Manager User Interfaces*

`quitWindowSystem`: optional
Quit the window system.
Parent: `windowManager`
Source: *SunView window system experience*

`realEstateMode`: alternative
Point to type.
Parent: `inputFocusSelection`
Source: *A Taxonomy of Window Manager User Interfaces*

`rectangularShape`: alternative

Windows must be rectangular.
Parent: `windowShape`
Source: *X10/uwm window manager experience*

`refreshAllWindowsOp`: optional
Redraw all windows.
Parent: `windowManager`
Source: *X10/uwm window manager experience*

`refreshWindowOp`: optional
Redraw the window.
Parent: `windowManager`
Source: *SunView window system experience*

`resizeIcon`: optional
Resize an icon.
Parent: `resizeWindowOp`
Source: *General*
Rules: requires `hasIcons`

`resizeInput`: mandatory
The way the user provides the resizing information.
Parent: `resizeWindowOp`
Source: *A Taxonomy of Window Manager User Interfaces*

`resizeWindowOp`: mandatory
Change the dimensions of a window.
Parent: `windowManager`
Source: *General*

`revertToNewFocus`: mandatory
The way a new input focus is chosen when the current one is deleted.
Parent: `listenerMode`
Source: *X11 Xlib documentation*

`rootNewFocus`: alternative
The input focus switches to the root.
Parent: `revertToNewFocus`
Source: *X11 Xlib documentation*

`scrollbarHighlight`: optional
Highlight the scroll bar.
Parent: `highlightedAreas`
Source: *Symbolics window system experience*

`selectionOrder`: mandatory
The order in which the command and the window are chosen.
Parent: `windowManager`
Source: *Open Look GUI documentation*

`specialAreaIconPlacement`: alternative

In a special area.
Parent: `newIconPlacement`
Source: *Macintosh window system experience*
Rules: requires `specialIconAreas`

`specialAreas`: optional
Reserved regions on the screen.
Parent: `windowManager`
Source: *A Taxonomy of Window Manager User Interfaces*

`specialAreasCoverable`: optional
Special areas may be covered.
Parent: `specialAreas`
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires `overlappedLayout`

`specialAreasRemovable`: optional
Special areas may be removed.
Parent: `specialAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`specialCommandAreas`: optional
Special areas for commands.
Parent: `specialAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`specialIconAreas`: optional
Special areas for icons.
Parent: `specialAreas`
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires `hasIcons`

`specialPromptInputAreas`: optional
Special areas for prompts and input.
Parent: `specialAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`stackingOrder`: mandatory
The stacking order for overlapping windows.
Parent: `overlappedLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`textCursorHighlight`: optional
Highlight the text cursor.
Parent: `highlightedAreas`
Source: *A Taxonomy of Window Manager User Interfaces*

`tiledArbitrary`: alternative
Window layout is tiled arbitrarily (windows not placed in columns).
Parent: `tiledLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`tiledColumns`: alternative
Window layout is tiled in columns.
Parent: `tiledLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`tiledLayout`: alternative
Tiled presentation.
Parent: `windowLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`titleBars`: optional
Title bars are used on windows.
Parent: `windowManager`
Source: *A Taxonomy of Window Manager User Interfaces*

`titlebarDeiconifiedIcons`: alternative
A windows title bar is its icon.
Parent: `deiconifiedIconDisplay`
Source: *Andrew window manager experience*
Rules: requires `titleBars`
requires `hasIcons`

`titlebarHighlight`: optional
Highlight the title bar.
Parent: `highlightedAreas`
Source: *A Taxonomy of Window Manager User Interfaces*
Rules: requires `titleBars`

`unchangedDeiconifiedIcons`: alternative
Icons remain visible as before.
Parent: `deiconifiedIconDisplay`
Source: *A Taxonomy of Window Manager User Interfaces*

`undoOp`: optional
Undo the last operation.
Parent: `windowManager`
Source: *A Taxonomy of Window Manager User Interfaces*

`unmappedDeiconifiedIcons`: alternative
Icons are made invisible by unmapping them.
Parent: `deiconifiedIconDisplay`
Source: *X10/uwm window manager experience*

`updateHiddenWindows`: optional
The ability to update windows which are hidden.
Parent: `overlappedLayout`
Source: *A Taxonomy of Window Manager User Interfaces*

`upperLeftCornerIconPlacement`: alternative
Upper left-hand corner.
Parent: `newIconPlacement`

Source: *X10/uwm window manager experience*

`warpToWindow`: optional
Change the current input focus to the deiconified window.
Parent: `deiconifyIconOp`
Source: *X11/twm documentation*
Rules: requires `changeFocusOp`

`windowConfiguration`: optional
Numerical feedback about the window's position, size, and shape.
Parent: `moveResizeFeedback`
Source: *X10/uwm window manager experience*

`windowLayout`: mandatory
The overriding display philosophy of the window manager.
Parent: `windowManager`
Source: *A Taxonomy of Window Manager User Interfaces*

`windowManager`: mandatory
The Window Manager
Parent: `top`
Source: *General*

`windowPositionIconPlacement`: alternative
The current window position.
Parent: `newIconPlacement`
Source: *SunView window system experience*

`windowShape`: mandatory
The allowable shapes for windows.
Parent: `windowManager`
Source: *NeWS window system documentation*

`zapEffectIcons`: alternative
Temporary lines flashed to show the results of icon operations.
Parent: `iconOperationFeedback`
Source: *X10/uwm window manager experience*
Rules: requires `hasIcons`

`zapEffectMoveResize`: optional
Lines flashed to show the results of move and resize operations.
Parent: `moveResizeFeedback`
Source: *X10/uwm window manager experience*
Rules: requires `ghostFeedback`
requires `eraseAfter`

`zoomEffectIcons`: alternative
A visual effect used to show the results of icon operations.
Parent: `iconOperationFeedback`
Source: *X11/twm documentation*
Rules: requires `hasIcons`

Appendix F: Hierarchical Window Manager Feature Listing

In the following listing indentation is used to show the hierarchical (parent/child) structure among the features. The numbers on the left indicate the indentation level. This diagram could be converted directly into the form of a feature diagram. To do so, the symbols should be interpreted as follows:

- Asterisks ("*") indicate *optional* features
- Bar ("|") indicates *alternative* features
- Dash ("-") indicates that the feature is applicable to this subtree
- All other features are *mandatory*

```
0  windowManager
1      createWindowOp
2      *  createIconified
1      destroyWindowOp
1      moveWindowOp
2      -  partiallyOffScreenWindows
2      -  zapEffectMoveResize
2      -  windowConfiguration
2      -  windowLayout
2      -  interactiveFeedback
2      *  moveIcon
2      *  constrainedMove
2      *  moveWindowColumnOp
2      *  abortMoveOp
2      *  moveErasure
3         |  eraseBefore
3         |  eraseAfter
2      *  exposeAfterMove
1      changeFocusOp
2      *  highlightInputFocus
3            highlightedAreas
4            *  borderHighlight
4            *  titlebarHighlight
4            *  textCursorHighlight
4            *  scrollbarHighlight
4            *  interiorHighlight
4            *  commandAreaHighlight
4            *  dimmedHighlight
3            highlightMethod
4            |  patternHighlight
4            |  colorHighlight
2      *  focusBeforeCommand
2      *  mandatoryFocus
2      *  multipleInputFoci
2      *  exposeAfterChangeFocus
1      resizeWindowOp
```

```
2      -  largerThanScreenWindows
2      -  interactiveFeedback
2      -  windowConfiguration
2      -  zapEffectMoveResize
2      *  resizeIcon
2         resizeInput
3         |  onePlaceResize
3         |  cornersAndHandlesResize
3         *  expandWindowOp
2      *  abortResizeOp
2      *  exposeAfterResize
1   *  hasIcons
2         iconifyWindowOp
3         -  zapEffectIcons
3         -  zoomEffectIcons
3         *  exposeAfterIconify
2         deiconifyIconOp
3         -  zapEffectIcons
3         -  zoomEffectIcons
3         *  warpToWindow
3         *  exposeAfterDeiconify
2      *  iconBox
2         iconUsage
3         |  processIcons
3         |  dataIcons
2         deiconifiedIconDisplay
3         |  unmappedDeiconifiedIcons
3         |  titlebarDeiconifiedIcons
3         |  dimmedDeiconifiedIcons
3         |  unchangedDeiconifiedIcons
3         |  iconBoxDeiconifiedIcons
2         iconIO
3         |  activeIcons
4            *  iconFocus
3         |  passiveIcons
2         newIconPlacement
3         |  upperLeftCornerIconPlacement
3         |  pointerPositionIconPlacement
3         |  windowPositionIconPlacement
3         |  openAreaIconPlacement
3         |  specialAreaIconPlacement
1      windowLayout
2      |  tiledLayout
3         |  tiledColumns
4            |  fixedNumberColumns
4            |  arbitraryNumberColumns
3         |  tiledArbitrary
2      |  overlappedLayout
3         *  partiallyOffScreenWindows
4            *  largerThanScreenWindows
3         *  updateHiddenWindows
```

```
4            *  hiddenInputFocus
3            stackingOrder
4               exposeWindowOp
4            *  hideWindowOp
4            *  circulateUpWindowsOp
4            *  circulateDownWindowsOp
1   *  refreshWindowOp
1   *  refreshAllWindowsOp
1   *  undoOp
1   *  quitWindowSystem
1      operationFeedback
2      *  iconOperationFeedback
3         |  zapEffectIcons
3         |  zoomEffectIcons
2         moveResizeFeedback
3         *  windowConfiguration
3         *  zapEffectMoveResize
3            interactiveFeedback
4            |  ghostFeedback
4            |  opaqueFeedback
1      windowShape
2      |  rectangularShape
2      |  arbitraryShape
1   *  specialAreas
2      *  specialPromptInputAreas
2      *  specialAreasCoverable
2      *  specialAreasRemovable
2      *  specialCommandAreas
2      *  specialIconAreas
1      inputFocusSelection
2      |  listenerMode
3            revertToNewFocus
4            |  parentNewFocus
4            |  rootNewFocus
4            |  nextOnStackingOrder
2      |  realEstateMode
1   *  titleBars
1   *  dropShadowsEffect
1   *  keyboardAccelerators
1   *  multipleObjectSelection
1      selectionOrder
2      |  objectAction
2      |  actionObject
```

Appendix G: Window Manager Entity Attributes

1. **Icon attributes:**
    - **Window Operations:**
        - *Deiconify*
2. **Main window attributes:**
    - **Window Operations:**
        - *Resize*
        - *Iconify*
        - *Expose*
        - *Hide*
        - *Circulate*
    - **Internals:**
        - Window gravity
3. **Pointer attributes:**
    - Position
4. **Screen attributes:**
    - Size
    - Shape
    - Resolution
5. **Window attributes:**
    - **Window Operations:**
        - *Create*
        - *Destroy*
        - *Move*
        - *Refresh*
    - **Internals:**
        - ID
        - Valid events (event suppression mask)
        - Parent window
        - ID of associated main window or icon
        - Application processes

- **Configuration/Geometry:**
    - Position
    - Shape
    - Size
    - Border width
    - Stacking order position
- **Visual attributes:**
    - Cursor picture
        - Source bitmap
        - Shape bitmap
        - Pair of colors
    - Mapping (mapped/unmapped)

References

[American 85] American Heritage.
The American Heritage Dictionary.
Houghton Mifflin, Boston, MA, 1985.

[Andribet 90] Pierre Andribet.
Cost Effectiveness of Using Ada in Air Traffic Control Systems.
In Barry Lynch (editor), *Ada: Experiences and Prospects- Proceedings of the Ada-Europe International Conference*, pages 11-23. Cambridge University Press, Cambridge, UK, June, 1990.

[Arango 88a] Guillermo F. Arango.
Domain Engineering for Software Reuse.
PhD thesis, University of California at Irvine, 1988.

[Arango 88b] Guillermo F. Arango.
Evaluation of a Reuse-Based Software Construction Technology.
In *Proceedings of the Second IEE/BCS Conference: Software Engineering 88*, pages 85-92. IEE, London, UK, July, 1988.

[Arango 88c] Guillermo F. Arango & Eiichi Teratsuji.
Notes on the Application of the COBWEB Clustering Function to the Identification of Patterns of Reuse.
Technical report ASE-RTP-87, ICS, University of California, Irvine, CA, July, 1988.

[Arango 89] Guillermo F. Arango.
Domain Analysis - From Art Form to Engineering Discipline.
In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 152-159. IEEE Computer Society, Washington, DC, May, 1989.

[Bailin 88] Sidney C. Bailin.
Semi-Automatic Development of Payload Operations Control Center Software.
Report prepared for NASA Goddard Space Flight Center, CTA Incorporated, Laurel, MD, October, 1988.

[Bailin 89] John M. Moore & Sidney C. Bailin.
The KAPTUR Environment: An Operations Concept.
Report prepared for NASA Goddard Space Flight Center, CTA Incorporated, Rockville, MD, June, 1989.

[Baldo 89] James Baldo Jr. (editor).
Reuse in Practice Workshop Summary.
Institute for Defense Analyses, Pittsburgh, PA, 1989.

[Barstow 85] David R. Barstow.
Domain-Specific Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1321-1336, November, 1985.
IEEE. Reprinted with permission.

[Batory 88a] Don S. Batory.
Building Blocks of Database Management Systems.
Technical report TR-87-23, University of Texas, Austin, TX, February, 1988.

[Batory 88b] Don S. Batory.
Concepts for a Database System Compiler.
Technical report TR-88-01, University of Texas, Austin, TX, January, 1988.

[Batory 88c] Don S. Batory, J. R. Barnett, J. Roy, B. C. Twichell & Jorge F. Garza.
Construction of File Management Systems from Software Components.
Technical report TR-88-36 REV, University of Texas, Austin, TX, October, 1988.

[Begeman 88] Michael L. Begeman & Jeff Conklin.
The Right Tool for the Job.
Byte 13(10):255-266, October, 1988.

[Biggerstaff 89a] Ted J. Biggerstaff.
Design Recovery for Maintenance and Reuse.
Computer 22(7):36-49, July, 1989.

[Biggerstaff 89b] Ted J. Biggerstaff.
Reuse and Design Recovery in MCC/STP.
Slides from Technical Presentation, Microelectronics and Computer Technology Corporation, Austin, TX, June, 1989.

[Booch 87] Grady Booch.
Software Components with Ada: Structures, Tools, and Subsystems.
Benjamin/Cummings, Menlo Park, CA, 1987.

[Borgida 84] Alexander Borgida, John Mylopoulos & Harry K. T. Wong .
Generalization/Specialization as a Basis for Software Specifications.
On Conceptual Modeling.
Springer-Verlag, New York, NY, 1984, pages 87-117.

[Bruns 88] Glenn Bruns & Colin Potts.
Domain Modeling Approaches to Software Development.
Technical report STP-186-88, Microelectronics and Computer Technology Corporation, Austin, TX, June, 1988.

[Chen 76] P. P. Chen.
The Entity-Relationship Model- Toward a Unified View of Data.
ACM Transactions on Database Systems 1(1):9-36, March, 1976.

[Coad 89] Peter Coad & Edward Yourdon.
Object-oriented Analysis.
Prentice Hall, Englewood Cliffs, NJ, 1989.

[Codd 70] E. F. Codd.
A Relational Model of Data for Large Shared Data Banks.
Communications of the ACM 13(6):377-387, June, 1970.

[Conklin 88] Jeff Conklin & Michael L. Begeman.
gIBIS: A Hypertext Tool for Exploratory Policy Discussion.
ACM Transactions on Office Information Systems 6(4):303-331, October, 1988.

[Cross 89] Nigel Cross.
Engineering Design Methods.
John Wiley & Sons, 1989.

[D'Ippolito 89] Richard S. D'Ippolito.
Using Models in Software Engineering.
In *Proceedings of Tri-Ada '89*, pages 256-265. Association for Computing Machinery, New York, NY, October, 1989.

[Firth 87] Robert Firth, William G. Wood, Richard D. Pethia, L. Roverts, et al.
A Classification Scheme for Software Development Methods.
Technical report CMU/SEI-87-TR-41 (ADA200606), Software Engineering Institute, Pittsburgh, PA, November, 1987.

[Gilroy 89] Kathleen A. Gilroy, Edward R. Comer, J. Kaye Grau & Patrick J. Merlet.
Impact of Domain Analysis on Reuse Methods.
Final Report C04-087LD-0001-00, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ, November, 1989.

[Goguen 84] Joseph A. Goguen.
Parameterized Programming.
IEEE Transactions on Software Engineering SE-10(5):528-543, September, 1984.

[Gomaa 84] Hasan Gomaa.
A Software Design Method for Real-Time Systems.
Communications of the ACM 27(9):938-949, September, 1984.

[Harel 89] David Harel & Sarah Rolph.
Modeling and Analyzing Complex Reactive Systems: The Statemate Approach.
In *Proceedings of the AIAA Computers in Aerospace VII Conference*, pages 239-246. The American Institute of Aeronautics and Astronautics, Washington, DC, October, 1989.

[Hess 90] James A. Hess, William E. Novak, Patrick C. Carroll, Sholom G. Cohen, Robert R. Holibaugh, Kyo C. Kang & A. Spencer Peterson.
A Domain Analysis Bibliography.
Special report CMU/SEI-90-SR-3, Software Engineering Institute, Pittsburgh, PA, July, 1990.

[IEEE 89] *Ada as a Program Design Language*
3rd revised edition, IEEE, Washington, DC, 1989.
ANSI/IEEE standard 990-1986.

[Jaworski 90] Allan Jaworski, Fred Hills, Thomas A. Durek, Stuart Faulk & John E. Gaffney.
A Domain Analysis Process.
Interim Report 90001-N (Version 01.00.03), Software Productivity Consortium, Herndon, VA, January, 1990.

[Lanergan 79] Robert G. Lanergan & Brian A. Poynton.
Reusable Code: The Application Development Technique of the Future.
In *Proceedings of the IBM SHARE/GUIDE Software Symposium*, pages 127-136. IBM, Monterey, CA, October, 1979.

[Lee 88] Kenneth J. Lee, et al.
An OOD Paradigm for Flight Simulators, 2nd Edition.
Technical report CMU/SEI-88-TR-30 (ADA204849), Software Engineering Institute, Pittsburgh, PA, September, 1988.

[Lubars 87] Mitchell D. Lubars.
Wide-Spectrum Support for Software Reusability.
In *Proceedings of the Workshop on Software Reusability and Maintainability*. National Institute of Software Quality and Productivity, Washington, DC, October, 1987.

[Lubars 88] Mitchell D. Lubars.
Domain Analysis and Domain Engineering in IDeA.
Technical report STP-295-88, Microelectronics and Computer Technology Corporation, Austin, TX, September, 1988.

[Maher 86] Mary Lou Maher & Panayiotis Longinos.
Development of an Expert System Shell for Engineering Design.
Carnegie Mellon University Engineering Design Research Center , Pittsburgh, PA, 1986.

[Matsumoto 84] Yoshihiro Matsumoto.
Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels.
IEEE Transactions on Software Engineering SE-10(5):502-513, September, 1984.

[McLeod 78] Dennis McLeod.
A Semantic Data Base Model and its Associated Structured User Interface.
PhD thesis, Massachusetts Institute of Technology, 1978.

[McNicholl 86] Daniel G. McNicholl, et al.
Common Ada Missile Packages (CAMP) - Volume I: Overview and Commonality Study Results.
Technical report AFATL-TR-85-93, McDonnell Douglas Astronautics Company, St. Louis, MO, May, 1986.

[McNicholl 88] Daniel G. McNicholl, Sholom G. Cohen, Constance Palmer, et al.
Common Ada Missile Packages - Phase 2 (CAMP-2) - Volume I: CAMP Parts and Parts Composition System.
Final Report AFAL-TR-88-62, Vol. I, McDonnell Douglas Astronautics Company, St. Louis, MO, November, 1988.

[Moore 89] John M. Moore, Richard Bentz, Manju Bewtra & Sidney C. Bailin.
Generic POCC Architectures.
Report prepared for NASA Goddard Space Flight Center, CTA Incorporated, Laurel, MD, April, 1989.

[Myers 88] Brad A. Myers.
A Taxonomy of Window Manager User Interfaces.
IEEE Transactions on Computer Graphics & Applications 8(5):65-84, September, 1988.
IEEE. Reprinted with permission.

[Nahaboo 89] Colas Nahaboo & Vania Joloboff.
GWM, The Generic X11 Window Manager.
In *Proceedings of XHIBITION '89*. Integrated Computer Solutions, Inc., June, 1989.

[Neighbors 80] James M. Neighbors.
Software Construction Using Components.
PhD thesis, University of California at Irvine, 1980.

[Neighbors 87] James M. Neighbors.
Report on the Domain Analysis Working Group Session.
In *Proceedings of the Workshop on Software Reuse*. Rocky Mountain Institute of Software Engineering, Boulder, CO, October, 1987.

[Peterson 86] James L. Peterson.
Design Issues for Window Managers.
Technical report STP-266-86, Microelectronics and Computer Technology Corporation, Austin, TX, August, 1986.

[Prieto-Diaz 87] Ruben Prieto-Diaz.
Domain Analysis for Reusability.
In *Proceedings of COMPSAC 87:The Eleventh Annual International Computer Software & Applications Conference*, pages 23-29. IEEE Computer Society, Washington, DC, October, 1987.

[Prieto-Diaz 90] Ruben Prieto-Diaz.
Domain Analysis: An Introduction.
ACM SIGSOFT Software Engineering Notes 15(2):47-54, April, 1990.

[Shlaer 90] Sally Shlaer & Stephen J. Mellor.
Recursive Design.
Computer Language 7(3):53-65, March, 1990.

[Wirth 71] Niklaus Wirth.
Program Development by Stepwise Refinement.
Communications of the ACM 14(4):221-227, April, 1971.

[Yourdon 78] Edward Yourdon and L. Constantine.
Structured Design.
Yourdon Press, New York, NY, 1978.

Table of Contents

List of Figures

List of Tables