A Software Tool for Specifying and Generating Displays in Object-Oriented Database Applications

Belinda B. Flynn
B.S. Electrical Engineering, University of Portland, 1985

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 1994

The dissertation "A Software Tool for Specifying and Generating Displays in Object-Oriented Database Applications" by Belinda B. Flynn has been examined and approved by the following Examination Committee:

Dr. David Maier
Professor
Thesis Research Adviser

Dr. David Novick
Assistant Professor

Dr. Robert G. Babb II
Associate Professor, University of Denver

Dr. T. Lougenia Anderson
Director of Database and Object Technology,
Sequent Computer Systems

Dedication

To my parents,

Antonia and Procopio Buenafe,

who provided for the educational foundation underlying this accomplishment.

Acknowledgements

My deep thanks to my adviser Dave Maier, who provided an invaluable contribution to this dissertation in the way of research ideas, managerial guidance and financial support. His commentary on earlier drafts of the dissertation led to large improvements in its content and readability. I also thank David Novick for help in developing the focus of my dissertation during his time as acting adviser. I am grateful to the thesis committee for their review of my work and their words of encouragement.

I would like to thank all who have contributed to the completion of this dissertation in some way, whether large or small. ParcPlace Portland allowed me use of their computer facilities and also provided a very helpful audience for practicing my oral defense. I have benefited from the company and friendship of many fine office-mates and colleauges at OGI. I greatly appreciate the moral support from all who gave it, with special mention to Lougie Anderson, Hitomi Ohkawa, Judy Cushing, and Adele Goldberg, who each provided me with an extra boost to keep going at some critical time. My family and friends supplied me with much needed stress relief and relaxation. Special thanks to the School of Traditional TaeKwonDo for helping me keep a healthy perspective on my work.

Dearest thanks to my husband Brian for being there with me through the best and worst of times as I pursued this degree.

Abstract

A Software Tool for Specifying and Generating Displays in Object-Oriented Database Applications

Belinda B. Flynn, Ph.D.
Oregon Graduate Institute of Science & Technology, 1994

Supervising Professor: Dr. David Maier

Database user interfaces are turning towards a style in which the main mode of interaction is editing and browsing data objects – giving the user the impression of directly affecting the data as concrete objects. To provide this feeling, the interface must be able to express the semantics and behavior of the underlying objects in a way that is intuitive for users. Several visual database interfaces have adopted this style to enable users to access databases more effectively without having extensive training or knowledge of the database schema.

Current database applications with visual interfaces receive little user-interface support from the database management system (DBMS). With record-oriented data models such as the network, hierarchical or relational models, the application is responsible for deriving the connectivity or object structure from data records. Thus the application, rather than the DBMS, is best suited to create and manage the object displays for the user interface. However, object-oriented DBMSs (OODBMSs) provide the ability to model data directly as structured objects. Since object structure is imposed by the database rather than the application, the database contains sufficient information for complex objects to be displayed directly, independently of the application. These new circumstances create an opportunity to factor another function out of the application: the management of object displays. A display facility associated with the OODBMS can take over this task, generating and managing interactive displays using only information stored in the database.

This dissertation discusses the design and implementation of the Object Display Definition System (ODDS), a prototype system created to investigate support for user-interface development in the context of OODBMSs.

Chapter 1

Introduction

As with user interfaces in general, database user interfaces are turning towards a style in which the main mode of interaction is editing and browsing data objects – giving the user the impression of directly affecting the data as concrete objects. To provide this feeling, the interface must be able to express the semantics and behavior of the underlying objects in a way that is intuitive for users. Several visual database interfaces [Goldman85, Bryce86, Leong89, Larson86] have adopted this style to enable users to access databases more effectively without extensive training or knowledge of the database schema. Domain-specific database applications most likely would gain similar benefits by having such interfaces. The overall objective of this research is the support of a graphical, interactive, user-interface style in applications that deal with complex database objects.

Current database applications with visual interfaces receive little user-interface support from the database management system (DBMS). With record-oriented data models such as the network, hierarchical or relational models, the application takes responsibility of displaying structured objects, since the database organizes data in "flat" generalized structures (e.g., relations). Consequently, the application must impose the connectivity or "object structure" on the data. Since the application builds structured objects from data records, it is best suited to create and manage the object displays for the user interface. The application is therefore performing two transformations: 1) translating between records and structured objects, and 2) translating between data objects and their display structure, or in other words, between an internal and an external representation. In addition, the application is typically responsible for maintaining integrity constraints concerning object structure.

Object-oriented DBMSs (OODBMSs) can model data directly as structured objects. Therefore, object construction need not be managed in applications and constraints regarding object structure can be associated with object classes. Since object structure is imposed by the

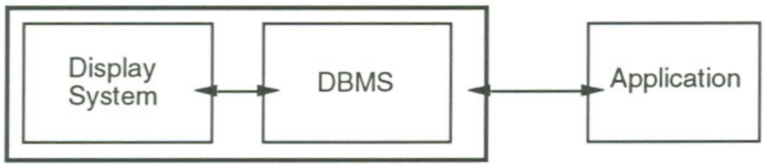


Figure 1.1: DBMS display facility

database rather than by the application, the database contains sufficient information for complex objects to be displayed directly, independently of the application. These new circumstances create an opportunity to factor another function out of the application: the management of object displays. If the DBMS manages object displays, the application only needs to specify how and when displays are created; it need not be involved with transformations between objects and their displays. A display facility associated with the OODBMS can take over this task, generating and managing interactive displays using only information stored in the database.

The database thus acts as a central resource for both the application program and the user interface (Figure 1.1). This arrangement can improve productivity due to the benefits of modularity:

- Display specifications can be developed more independently of application programs. Once some basic requirements are established, development of the application programs and their associated displays may be done in parallel.
- Displays are reusable among different applications that operate on the same database.
- Display implementation is less tightly coupled with the rest of application processing, facilitating reuse of display components. When the two are tightly coupled, it is difficult to know exactly what parts of the code to extract to capture particular subparts or functionalities within a display.
- It is easier to provide multiple user interfaces and experiment with alternative displays for an application.

In summary, the overall goal is to provide support for producing interactive displays of complex database objects. The general approach of this research is to move display management from the application to a display system that directly accesses database objects. The basic motivations are to utilize advantages offered by OODB technology, to improve productivity in creating displays through modularity, and to increase the capabilities in user-interface tools for reflecting the state and semantics of objects being displayed. To investigate this approach, I have developed a prototype system called the Object Display Definition System (ODDS).

1.1 Related Work

This section presents an overview of existing approaches to support display development for complex database objects. Many database systems include application development tools (sometimes called 4th Generation Languages) that support the creation of form-based user interfaces. These tools generally provide a set of display building blocks, such as text-entry fields, radio buttons and menus, that can be included in forms and used for entering pieces of data or invoking database operations or queries. An example is the Forms Application Development System (FADS) [Rowe82] for Ingres databases. In FADS, a database application consists of a collection of frames, where a frame contains a menu of operations and a form in which data is entered and displayed. FADS provides generic frame operations such as movement between frames and help commands, and application-specific operations are written using a QUEL-like language. The Picasso system [Rowe91] is based on the concepts of FADS, and used with the Postgres DBMS [Stonebraker86]. Whereas FADS was developed for alphanumeric terminals, Picasso uses graphical user interface (GUI) features, e.g., multiple windows and bitmapped graphics, in its development environment and in the database applications developed. Picasso also adds the ability to run an application and edit its specification without having to restart it.

Another area of related work involves display generation tools that produce browsers for object-oriented databases. These systems generate default displays having a standard format, based on the structure information kept in the object classes. The generated displays include predefined techniques for browsing, such as switching between displays with different levels of detail. Some systems allow the designer limited ability to customize the default displays through a language or graphics editor. Examples include LOOKS [Deux91], which is part of the programming environment for the O2 OODBMS, and KIVIEW [Laenens89], used with the KIWI OODBMS. The Smalltalk Interaction Generator (SIG) [Maier86] is a prototype system that also produces displays with browser capabilities, but generates them from a display description that a designer creates from scratch or by modifying an existing description. Multiple display descriptions can be associated with an object class, and a description can define alternative display arrangements that are invoked depending on a displayed object's state. These systems are described in more detail in Section 2.3.

Another approach for supporting display creation is to integrate a class library or user-interface construction tool with the programming facilities of an object-oriented DBMS. For example, the ETDB++ system [Schmidt90] supports development of a database application, including its user interface, through a generic application class library (GACL). The combination of certain classes in the GACL forms a framework that provides many of the functions commonly found in database applications. ETDB++ seeks to support GUIs using direct manipulation (e.g., a graphical editor); the functions provided include event management, window operations such as moving and resizing, a set of predefined dialog components, and a foundation for reversible commands. Another example is FaceKit [King89], a tool for building database user interfaces in the Cactis OODBMS. FaceKit has knowledge of database schemas and the data manipulation facilities available in Cactis. Thus, it can supports creating graph diagrams or forms to present a database schema or object, and defining the operations invoked through the user interface. FaceKit stores the interface definitions in the database as methods.

Several core concepts in the design of ODDS come from the PROTEUS project [Anderson86], which sought to promote reusability by storing display specifications, as well as schemas and queries, as objects in the database. Storing this meta-information in the database also has the advantage that it is placed under programmer control and may be examined, modified, and displayed just as other objects are.

Research on user-interface tools that are not specifically aimed at database applications is also relevant to this work; this research is discussed in the following section.

1.2 Semantic Feedback Support Issues

This section discusses the current approaches to support semantic feedback in user-interface construction tools. *Semantic feedback* refers to changes in the display images (or sounds) that inform the user about the state of application objects in response to a user's action upon those objects. It can also include providing error or help messages about the application domain, such as the type of object expected for a particular entry field. Semantic feedback is essential to creating the impression that the display images actually are the objects they represent, and that the user is working directly with those objects. The implementation of semantic feedback requires knowledge of the application semantics to be somehow accessible to the user interface, either included in the user interface initially or communicated to it during execution. This requirement has raised questions about the proper runtime architecture for user-interface tools, particularly regarding how the constructed user interface interacts with the application it serves.

A substantial amount of research has been directed towards software tools called User Interface Management Systems (UIMSs), whose goal is to construct user interfaces automatically from a declarative, high-level specification created through a language or a direct-manipulation environment. A fundamental concept in the use of UIMSs is that they separate the user interface from the rest of the application, just as a DBMS factors out data management functions. The advantages of this separation are similar to the benefits of modularity as described in the previous section, although the context is slightly different – the benefits apply to the whole user interface, rather than just to the individual displays that compose the user interface.

In many UIMSs, the separation between user interface and application is implemented based on the Seeheim model, which has been used both as a guideline for organizing the contents of specifications *and* as a model for the runtime operation of the user interface and application. The Seeheim model (Figure 1.2a) identifies three logical components in the user interface [Green85a]:

1. The *presentation component* draws the screen images and reads physical input devices, converting the input data into the more abstract data structures required by other components.

2. The *dialogue control component* basically translates user actions into calls to application routines. In the opposite direction, it translates data and actions from the application to appropriate actions by the presentation component.

3. The *application interface model* is a representation of what the user interface knows about the application, such as the application routines available and constraints on how to use the routines.

The system architectures for many Seeheim-based UIMSs have the user interface and the application operating mostly independently of each other and communicating over a low bandwidth connection, such as procedure calls. However, researchers have recognized that this type of architecture is not suited to supporting semantic feedback [Myers89]. Incorporating the

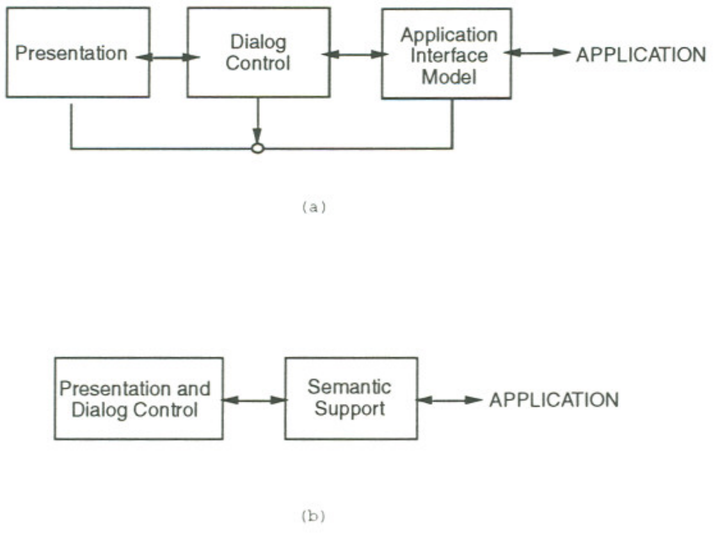


Figure 1.2: UIMS Architectures

necessary application information into the user-interface specification means that the same information exists in two places, which works against modularity: if there is some design change in the application, it must also be changed in the user-interface specification. Communicating the necessary information between the two components on demand is not acceptable with a low-bandwidth connection, because frequent communication would cause a noticeable delay in the user interface's operation.

Some have suggested that these difficulties with supporting semantic feedback indicate that the basic principles of UIMSs are not valid for direct-manipulation interfaces [Rosenberg88, Linton89]. One direction of research forgoes the separation originally put forth and seeks to modularize the application in other ways. For example, the toolkit or class library approach is being investigated as a more suitable means of producing user-interfaces that require semantic feedback. This alternate approach gives each object the ability to display itself, thus integrating the object being displayed and the data structures needed to display it.

The direction taken in this dissertation is to pursue further the principle of separating data manipulation from data presentation to obtain the benefits offered by that modularity. To achieve those benefits, it is more important to maintain the separation at the specification level, rather than in the runtime operation of the user-interface and application. In other words, the Seeheim model may be appropriate as a logical model of the functions that must be present in the runtime system of a UIMS, but it need not dictate the actual system architecture.

An alternate system architecture (Figure 1.2b) has been used in several UIMSs specifically aimed at supporting direct-manipulation interfaces [Hudson87]. The most significant difference in this architecture is that the user interface and application communicate through shared access to both the data objects being displayed and information about the objects' structure and activities. In effect, data that was available only to the application in other UIMS architectures is made accessible to the user-interface. With this mechanism, the user interface no longer has to call application-supplied procedures to update the displayed objects or request information about them. Instead it can operate on or query the objects directly. Thus, the central data store itself is a channel connecting the user-interface and application, and in a sense, provides a higher bandwidth of communication between them than does the use of procedure calls. The systems based on this architecture are categorized as *data-oriented* UIMSs. Examples include HIGGENS [Hudson88] and the Serpent UIMS [Bass90].

This alternate architecture coincides with the architecture described in the introduction, which is made feasible by OODB technology. Using this architecture imposes some requirements on application design. It is necessary to distinguish the data to be displayed and the information needed to produce semantic feedback, and they must be entered into the central data management component. Since the application is no longer solely responsible for the changes in the objects that it manipulates, it must be prepared to recognize and handle changes made to them by some other agent. These requirements are already inherent in designing database object-oriented applications.

1.3 Contributions of this Research

The central question addressed in this dissertation is how a display generation tool can accommodate a separation between the specification of the displays and the application program, while supporting the specification and execution of semantic feedback. The basic approach of data-oriented UIMSs, which is to make the semantic information available to the constructed user interface through a central data management component, is also the basis for the solution pursued in this research. My research further develops this basic approach, seeking to define an information base for the central data store that is richer than those used in existing data-oriented UIMSs and, consequently, to extend the specification capabilities found in those systems.

One contribution of this research is the design of a system architecture in which an OODBMS serves the role of the central data manager that provides an information channel between the user-interface and the rest of the application. The design details of the architecture clarify which facilities are required for the OODBMS to perform that function. Furthermore, the design characterizes the types of semantic information that should be present in the central data store to support certain kinds of semantic feedback.

To express the actions that constitute semantic feedback, it is necessary to specify how the display changes in response to changes in the underlying objects. Another contribution of this research is the design of a specification framework that allows the expression of a variety of translations from data objects to their external representations (i.e., display images) beyond what is expressible in existing tools. The extension of specification capabilities is made possible by the semantic information that is available in the OODB both at design time and run time.

A key difference between the design of ODDS and that of existing data-oriented UIMSs is that the display descriptions are placed in the central store along with objects being displayed, and the descriptions take the form of complex objects. Using objects as the form for specifications emphasizes the division between specifying object displays and implementing the rest of application. The specification objects are distinct entities from the database class methods and other program code used to describe the application's functionality. Thus, in addition to extended expressiveness, another contribution related to the specification framework is the data model designed to capture display functionality as complex objects.

The object form of specifications also facilitates reuse because a specification object can easily be copied for use in another specification, or several specifications can reference a particular specification object as a subpart. Capturing specifications as database objects means that they are centrally available, and can be used by multiple applications. Furthermore, for DBMSs that support several application languages, use of the display services is not tied to a particular language because the usage only requires access to the database rather than linking to a code library.

This dissertation does not claim to define a particular user-interface style that is especially suitable for interacting with OODBs. Rather, the main focus of the research is on providing support for producing displays more efficiently, while allowing for variety in the display behavior and appearance. The research does not deal with human factors issues regarding appropriate features for the content of the displays. Another related area not addressed is producing default displays for objects based on their class definitions – the system does not decide how displays look or behave, rather it relies on the display designer to create and place specifications in the database. However, this research provides a good foundation for future work on describing the derivation of a display specification object from a class-defining object.

1.4 Example Database and Displays

This section introduces a sample database that will be a source for examples and illustrations throughout the dissertation. References to object classes and individual objects are distinguished by their font, e.g., '`Recipe`' denotes a class and 'Recipe' denotes an object. Displays for this database were created to test ODDS's prototype implementation. The database holds nutritional information for foods, recipes, and diets, as shown by the class descriptions and sample objects in Figures 1.3 and 1.4.

An instance of the `Food` class has a name and holds information about the nutrients it contains. That information is contained in a referenced object, of class `NutritionLog`. A NutritionLog specifies the amounts in grams of protein, fat, and carbohydrate for a serving of a Food. `ExtNutritionLog` (extended nutritionLog) is a subtype of `NutritionLog`, and its instances hold additional information on the amounts of vitamins in Foods.

The class `Recipe` is a subtype of `Food`, so a Recipe includes all the information found in a Food. In addition, a Recipe includes a preparation time (in minutes), the number of servings the Recipe makes, a list of ingredients, and a list of instructions. The ingredients of a Recipe are instances of `RecipeItem`, which includes the amount of an ingredient and the ingredient itself. Note that an ingredient may be either a Food or a Recipe because of the subtype relationship. A RecipeItem may also specify a particular form of an ingredient using a text string such as 'chopped' or 'diced.' The ingredient amount is defined in a `Measurement` object consisting of a quantity and an instance of `MeasureUnit`, which is a subtype of `String`. A fixed set of MeasureUnits are defined and stored as a part of class initialization; thus `MeasureUnit` serves as an enumeration type. The amount of protein, fat, or carbohydrates in a Recipe can be derived by taking the sum of the ingredients' nutrient values and dividing by the number of servings.

An instance of `Diet` specifies the name of the person who is following the Diet and a minimum and maximum daily amount for each of the nutrients; this information is stored as two NutritionLog objects. It also includes a schedule represented as a list of `DayPlan` instances,

```
class Food
  name: String
  nutrients: NutritionLog


class NutritionLog
   protein: Real
   fat: Real
   carbohydrates: Real


class ExtNutritionLog: subtype
                 of NutritionLog
   vA: Real
   vC:  Real
   vB12:  Real
   vD:  Real
```

```
class Diet
   owner: Text
   disallowed: List of Food
   dailyNeeds: NutritionLog
   dailyLimits: NutritionLog
   schedule: List of DayPlan
   currentNutrition: NutritionLog

class DayPlan
   breakfast: Meal
   lunch: Meal
   dinner: Meal

class Meal
   dishes: List of Food
   nutrients: NutritionLog
```

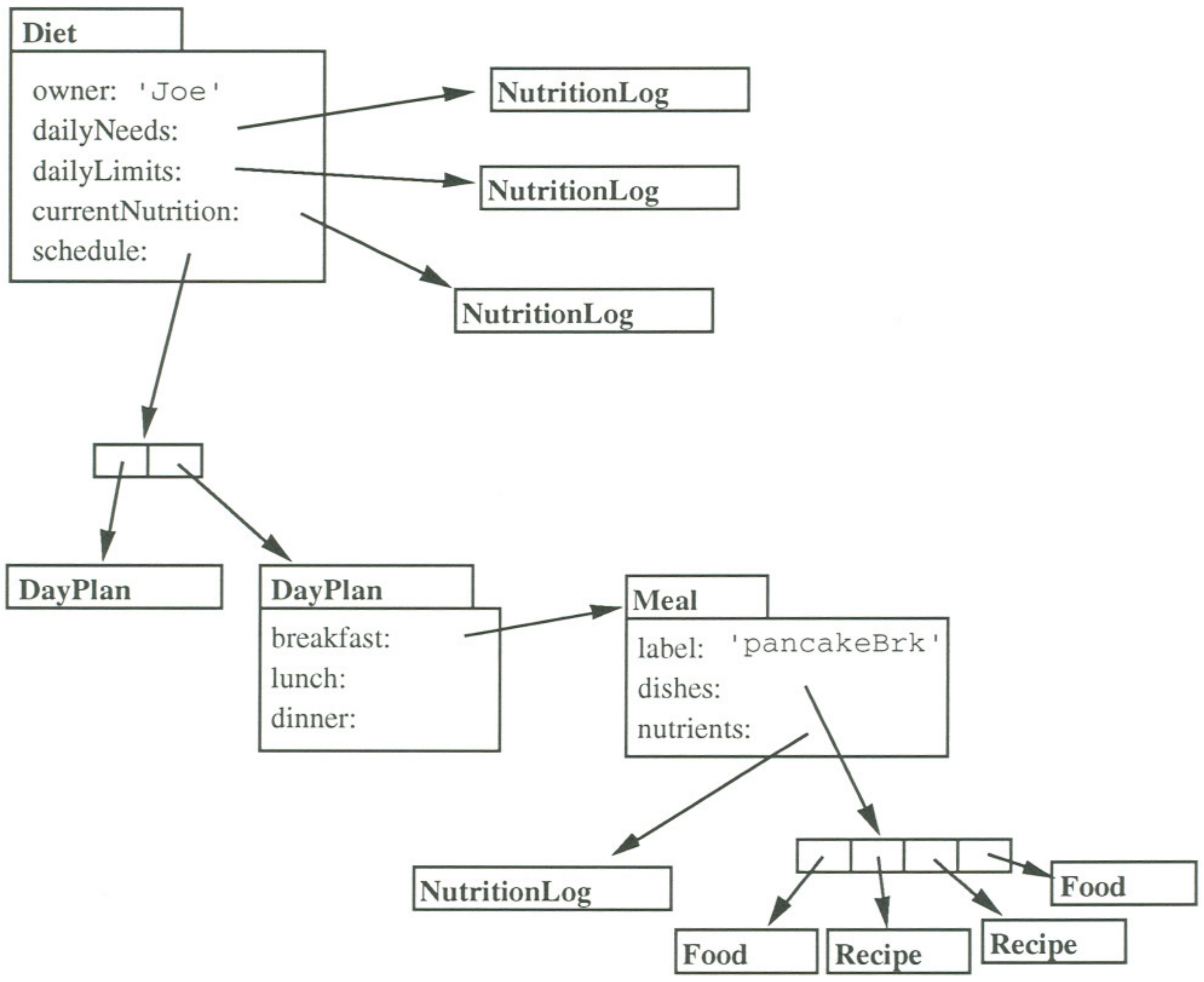


Figure 1.3: Schema and Instances of Foods and Diets

```
class Recipe:  subtype of Food
   prepTime: Int
   numServings: Int
   instructions: List of Text
   ingredients: List of RecipeItem
```

```
class RecipeItem
   amount: Measurement
   form: Text
   ingredient: Food

class Measurement
   quantity: Real
   measure: MeasureUnit
```

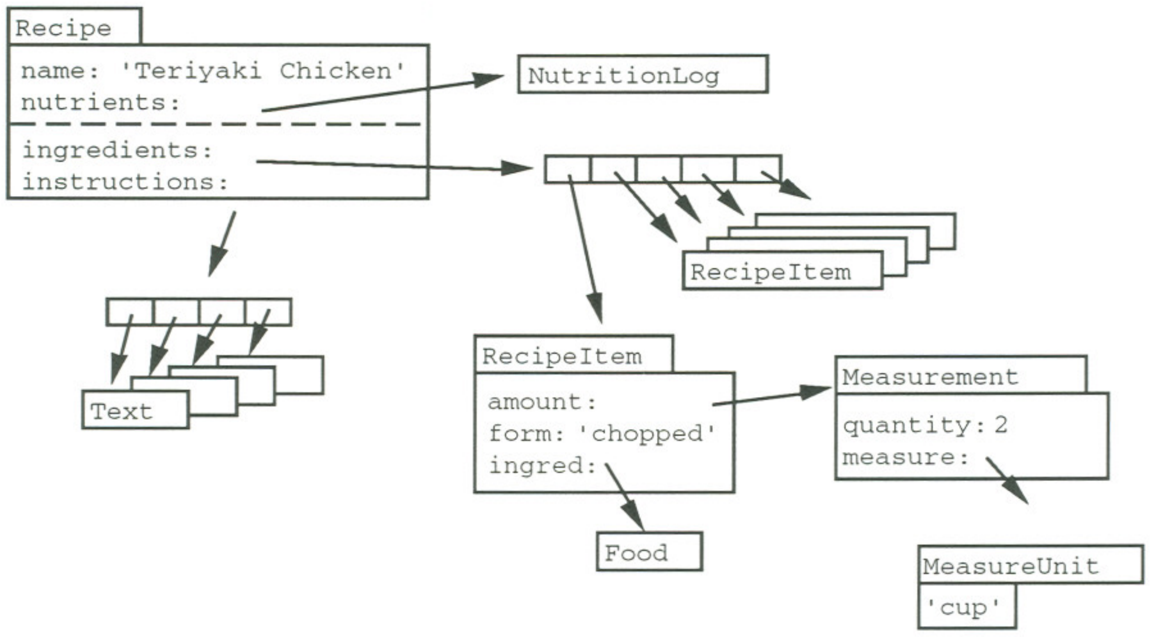


Figure 1.4: Schema and Objects in Recipes

each of which holds three Meal objects: breakfast, lunch, and dinner. An instance of `Meal` includes a list of dishes, which are Food objects. It also includes the Meal's nutritional content, calculated from the NutritionLogs in the Foods. A Diet object must keep track of the changing amounts of nutrients as Meals are added and deleted in modifying the diet schedule.[1] The current value for the daily average of each nutrient is also stored.

A DayPlan does not store a NutritionLog object. Rather, the message'nutrients' compute a DayPlan's nutritional content from the Meals it contains. A DayPlan's nutrition values are affected whenever its breakfast, lunch, or dinner are changed. Rather than updating a NutritionLog each time a Meal changes, the nutrient values are re-calculated only when requested by an application or another database object. Similarly, the daily average of nutrients for a Diet is affected whenever a DayPlan is added to the schedule or when the nutrition of an existing DayPlan changes. Unlike a DayPlan, a Diet does store a NutritionLog object. The schema was designed this way to show how different implementation possibilities are handled when specifying displays.

Aside from the class definitions above, the database also defines several global variables. The variables FoodData and MealData hold sets of Foods and Meals, respectively. The variable BreakfastChoices holds an array of the Meals that are allowed as the breakfast of a DayPlan. Similarly, the variables LunchChoices and DinnerChoices hold allowable Meals for lunch and dinner, respectively. `Diet`'s class definition includes the messages 'getBreakfasts', 'getLunches' and 'getDinners' that each return the array object in the appropriate global variable.

Throughout the dissertation, several terms are used regarding object modelling. An object's definition, or *type*, lists an object's attributes and its relationships to other objects. An object of type X has an *abstract state* consisting of the data associated with those attributes and relationships; i.e., the abstract state consists of the *attribute values* and *composition* of the object.

An attribute value may be either a basic value such as a character, or a structured value holding basic values, e.g., a string, which is an array of characters. An attribute value is atomic, meaning that its components by themselves are not significant to the domain being modeled. For example, the characters within a string are not significant entities with respect to a database domain of foods and diets. An object's composition consists of its connections to other objects,

[1]In my usage of the prototype database, a Meal is treated as a fixed unit and the Foods of a Meal were not modifiable through the displays.

where the connection represents a semantic relationship; e.g., a DayPlan object references a Meal to represent the "breakfast" relationship.

The following chapter elaborates on the display qualities that this research seeks to support. These qualities motivate several basic choices in ODDS's design. Chapter 3 discusses design objectives that follow from the choices described in Chapter 2. Chapters 4 through 7 then describe display specifications created in ODDS, how the system is used by an application program, general activities in display generation and execution, and key issues in the implementation of those activities. Chapter 8 describes displays that were constructed using ODDS, and presents some evaluation of the ODDS prototype with respect to its expressiveness and runtime performance. Chapter 9 concludes the dissertation, summarizing the work accomplished and opportunities for further research.

Chapter 2

Critical Design Aspects

2.1 Design Alternatives and Desired Choices

This section discusses three design aspects that significantly affect support for semantic feedback while keeping display specifications separate from the application program. These aspects are: the database model, the degree of display behavior capture in descriptions, and the level of support for describing display responsiveness to changes in underlying data.

The first design aspect is the database model for defining object structure and behavior. The database model defines how much of the objects' semantics can be maintained in the data management component. The level of semantic information in turn determines how much information can be provided to displays at runtime to reflect object state. The second design aspect is the extent of display behavior captured in display definitions. Any display behavior not expressed in the display definition must be implemented by the application program. Thus, this aspect directly affects the degree of logical separation of display management from the application. The third aspect of concern is the system's ability to specify and produce behavior in which the display's format is affected by changes in the connections between objects. This ability is necessary for presenting the semantics and activities of objects with dynamically changing complex structure. The three aspects are discussed in the following subsections.

2.1.1 Database Model

Although all OODBs by definition support complex objects [Zdonik90], they differ in the extent to which object behavior can be defined. Some database models, called *semantic database models* [Hull87], concentrate only on structural abstractions, e.g., aggregation, set grouping, and relationships. Other models provide for the expression of rules or constraints concerning their objects' semantics. One kind of rule describes the creation of *derived data* computed from existing data. Other rules are one-way constraints that state how changing a particular data item will cause values in other data items to be re-evaluated. These rules produce the notion of *active data* that can react to changes. Certain aspects of the objects' behavior can be described using these techniques; however, some systems using such models, e.g., the Cactis [HudKing87] and HiPAC [McCarthy89] DBMSs, do not support rules involving an object's connections to other objects.

Behavioral models use the concept of abstract data types to associate general behavior with complex objects. The GemStone Object DBMS [Butterworth91, Maier87] and the O2 System [Deux91] are examples of systems using behavioral models. An abstract data type includes a *protocol* that defines how one may interact with objects of that type; the protocol consists of a set of *messages* that may be used to query or manipulate the object's state. An abstract data type also defines an internal representation, which may be a complex structure, and the implementation, or *method*, for each message in the protocol. Behavioral models enforce a strong view of encapsulation, meaning that the messages of a type do not necessarily allow direct access to an object's internal representation. An object type as described in Section 1.4 is equivalent to a message protocol of an abstract type.

The choice of database model affects the capabilities of the display system because it determines how much state information the DBMS can provide to the display system. In particular, the display system should not be restricted to queries about attribute values, since information on object composition is also relevant to producing semantic feedback. Choosing a behavioral database model provides the advantage that the information regarding the objects' integrity constraints and behavior can all be managed within the database. In non-behavioral models, this management must be split between the database and the applications that use the data. In addition to complicating matters for the display system, the split introduces the possibility that one application might violate constraints being maintained by another application.

Another advantage of using a behavioral database model is that the message paradigm provides a mechanism for the display system to manipulate database objects. Within a display description, one may specify that some message should be sent to a database object after a particular event or event sequence has occurred. Consequently, the application program is not involved in every update of the database objects; the display system may perform some updates directly on behalf of the user.

2.1.2 Scope of Display Descriptions

Different tools for building user interfaces capture various levels of display activity. At the lowest level, toolkits provide predefined components such as buttons, menus, and gauges, and require that the designer build up displays by using a procedural programming language to combine the components. Toolkits give no support for logically separating application processing from the code that manages the displays; rather, display code is interspersed with the rest of the application code.

UIMSs capture more activity than toolkits since the designer can specify how the components are composed. The display descriptions are defined using a special-purpose language or environment and are thus separate from the application code. As mentioned in Section 1.2, many UIMSs that are based on the Seeheim architecture separate the user interface and application in such a way that the user interface is concerned only with interpreting user input, providing feedback as the input is parsed, and sending an appropriate request to the application. Only the application has direct access to its objects; there is no explicit means for the user interface to find out how the application's objects are structured or to invoke actions on them without involving the application.

The highest level of support captures display behavior that reflects how the underlying objects have changed. Basic display responses include feedback about attribute values, which is determined by a simple mapping from the database value to a graphical object. More complex responses involve some interpretation or intermediate processing of the updated values in the underlying objects. For example, a display change may be triggered by certain conditions in object state, such as a list being empty or non-empty. Another case would be a display change that occurs in addition to feedback on data values. For example, when adding a Meal to a Diet's schedule causes the Diet's fat content to exceed the set limit, the display might reflect the assignment, but also bring up a notifier stating that a limit has been violated. The specification of these kinds of display responses requires a way to refer to the attributes and relationships of a displayed object (so that values or connections can be queried). Data-oriented UIMSs have addressed this level of support by introducing a data model in which the objects' definitions are available for use in describing displays. This highest level of support, the description of basic *and* complex display feedback, is an objective for ODDS.

2.1.3 Display Responsiveness

The third design aspect is the system's ability to produce displays whose format can change and be responsive to changes in underlying objects. Some systems support *static* representation, where displays are limited to a single fixed template for all instances of a class and only the basic data values in the display may change as it executes. Some systems allow *multiple* representation, so that an instance's display can have differing formats depending *only* on conditions evaluated at the time the display is created. *Dynamic* representation allows displays whose format may change during their lifetime, to reflect the state of the underlying object, to respond to some user request, or to meet some space requirements. A *format change* in a display means altering the number or arrangement of display subcomponents, the graphics connecting subcomponents, or the format of any subcomponent.

Dynamic representation is important for supporting semantic feedback because displays of complex objects should reflect changes in an object's composition as well as changes in its attribute values. Composition changes that create or delete the connections of an object could add or delete the subcomponents of a display. In addition, replacing one of the object's references can change the format of a display subcomponent. Such a change occurs if the newly referenced object has a different type from the previously referenced object. In many OODB models, the referenced object in a particular relationship is declared to be of a specific type. However, the actual object referenced can be an instance of the declared type or one of its subtypes, which may have different display requirements. For example, an ExtNutritionLog holds extra information on vitamins, so a display for `ExtNutritionLog` may present information that is not present in `NutritionLog`.

In particular, the ability to reflect object composition is useful in database browsers that facilitate exploring, querying, and manipulating the database contents. For example, in the browsers generated by SIG [Maier86] and LOOKS [Deux91], referenced objects are displayed using nested displays, so compositional changes are shown when viewing data. Many browsers allow the user to control the level of detail at which objects are viewed. This "zooming" operation is another form of dynamic representation, although it is not tied to changes in the state of the object being displayed.

To summarize, the system features desired for supporting interactive displays are:

- a behavioral model for defining complex objects

- support for capturing display behavior that includes complex responses to database changes
- support for dynamic representation

2.2 Suitability for Direct Manipulation

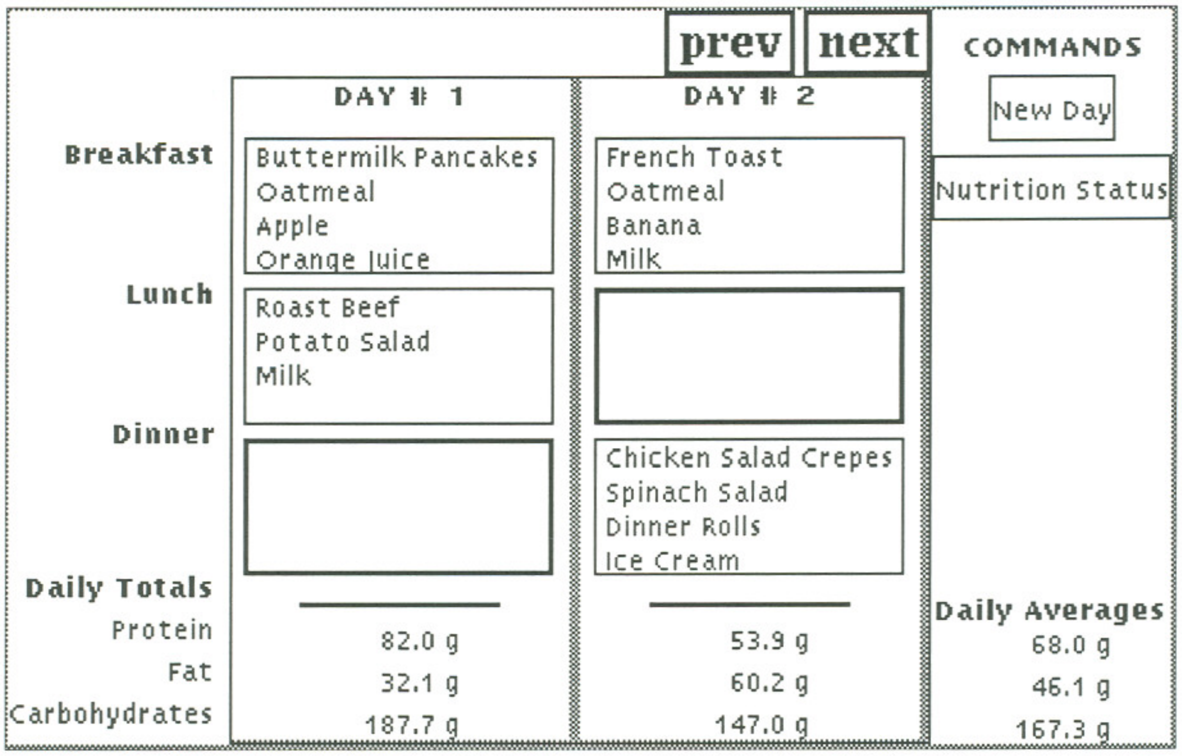
2.2.1 Aspects of Direct Manipulation

As mentioned earlier, one main objective is to support displays that provide a sense of direct interaction with the database objects of interest. Studies of direct-manipulation interfaces identify two factors that are essential to obtaining this feeling of directness in user interfaces: *direct engagement* and *reduction of distance* [Hutchins86]. An understanding of these factors is a basis for seeing how the three desired features of the previous section contribute to display systems that support direct manipulation. The following discussion summarizes the main concepts of the two factors, as developed in that research.

Direct engagement is the sense that the displays are active representations of the underlying objects and that the user is controlling the displayed objects through his or her physical actions. In interfaces without direct engagement, the user is provided with a language to specify abstractly how the objects of interest should be manipulated, and those objects are displayed only when specifically requested; the metaphor for interaction is that the user and the computer system are having a conversation about an implicit set of objects. In contrast, direct engagement requires a "model world" metaphor, where the interface embodies a world in which the objects of interest remain visible as the user works with them, making them an explicit set of objects. Furthermore, the user performs actions to affect the objects' appearances.

Direct engagement is produced by displays that exhibit behavior of their own, rather than being merely static output printed on the screen. Direct engagement also requires a special relationship between input and output called *inter-referential I/O* [Hutchins86], in which output (i.e., the display images) may serve as components of input expressions. In other words, selecting or referencing an object's display is part of an input sequence that invokes an operation on the object. Since the input sequence results in changes to the display image, it appears to the user that input directed towards an object evokes its behavior.

For example, the diet display image in Figure 2.1 presents information and is also available as the subject of subsequent input expressions that invoke operations on the schedule. To add another day to a schedule, the user would select this display as the current subject of operation



	DAY # 1	DAY # 2	Daily Averages
Breakfast	Buttermilk Pancakes Oatmeal Apple Orange Juice	French Toast Oatmeal Banana Milk	
Lunch	Roast Beef Potato Salad Milk		
Dinner		Chicken Salad Crepes Spinach Salad Dinner Rolls Ice Cream	
Daily Totals			
Protein	82.0 g	53.9 g	68.0 g
Fat	32.1 g	60.2 g	46.1 g
Carbohydrates	187.7 g	147.0 g	167.3 g

Figure 2.1: Display of Diet Schedule

and execute the *Add Day* command. The display would then change to show another column for the added day.

The other factor contributing to directness is the reduction of distance, where distance refers to the relationship between a user's task and the interface's mechanisms for performing that task. Thus, the notion of distance emphasizes that the directness of a display is always relative to a particular task. The distance between a task and a user interface has two parts. First, *semantic distance* indicates how closely the task is matched by the commands and the kinds of display feedback chosen for the interface. With respect to input, semantic distance is bridged by choosing commands that allow the user to accomplish a task in a concise manner. With respect to output, it is bridged by providing feedback information that allows the user to evaluate readily whether the desired goal is being achieved. In other words, semantic distance indicates the directness of the *meanings* behind the input and output expressions that occur through a user interface. The meaning behind an input expression is the operation that is invoked when the interface receives the sequence of inputs in that expression. The meaning behind an output expression is a directive about what information to present or change in the display image, such as "'show that an item has been added to this list."

Second, *articulatory distance* indicates how well the those meanings (for both input and output) are expressed by physical forms. Physical forms for input expressions include sequences of mouse movements, mouse button clicks, and key presses. For output expressions, they include character strings, bitmap images, sounds, or compositions of several output forms. Articulatory distance is reduced by choosing the appropriate physical forms for requesting a given operation and for representing the concepts and objects of the application domain. Overall, a reduced distance means that, for a given user task, the user can easily query or process the objects as desired and can get immediate feedback to evaluate the results of his or her actions. Since this effort is reduced, the interaction feels more direct with respect to the user task.

Some examples of user-interface features may help to clarify the concept of distance. Suppose the user wants to find foods whose protein content falls within a certain range. If a recipe browser provides only a search command where the user enters a single value for a nutrient as the search criteria, it has a large semantic distance relative to the task because the user must perform the search for each value within the desired range. Considering semantic distance for output, suppose a user wishes to keep the average daily amount of fat within a certain range as a Diet's schedule is being updated. The schedule display in Figure 2.1 reduces semantic distance because it shows the average for each nutrient in addition to the daily values; otherwise, the user would have to calculate the averages to accomplish the task. The display in Figure 2.2 further reduces semantic distance, since it presents the desired range for each nutrient and provides feedback on whether an acceptable level has been achieved.

To understand articulatory distance, consider two possible forms for specifying a breakfast connection between a DayPlan and a Meal in the display in Figure 2.3. One possible form for this operation would be to tab through the column of DayPlans, press the return key to select one, and select a meal similarly. A form with less articulatory distance would be to select the DayPlan and Meal by pointing and clicking with a mouse.

2.2.2 Support for Directness

The three design features from the previous section are needed to support the specification of display features that foster direct engagement and the reduction of distance.

The use of a behavioral model contributes to direct engagement because it can supply a more complete account of the objects' behaviors, including information on changes to an

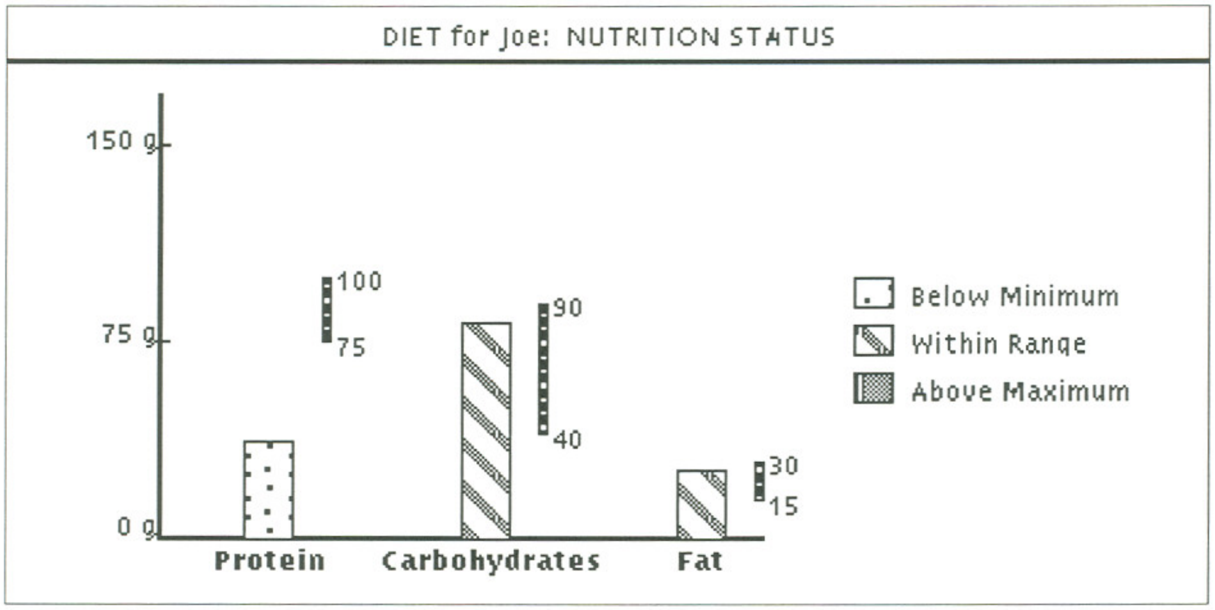


Figure 2.2: Display of Nutrition Status

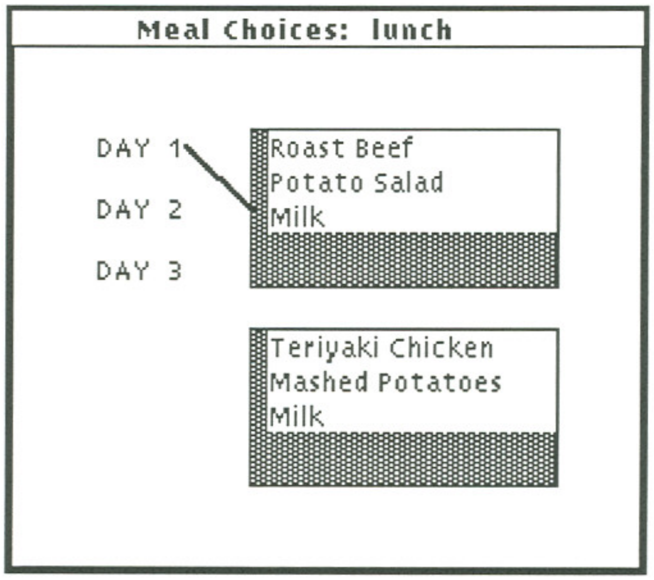


Figure 2.3: Display for Choosing Predefined Meals for a Diet Schedule.

object's composition. Reducing semantic distance in a display's output often requires presenting information that is not initially part of any object's state, but is computed from state values of an object or group of objects. The message paradigm in behavioral database models can be used to easily extend the information on object state as necessary.

Supporting dynamic representation in a user-interface tool helps produce direct engagement because it implements an important form of display behavior. The display can behave similarly to the objects it represents by changing format in accordance with changes in the connections among the underlying database objects. This support also helps reduce articulatory distance when the intent of output expressions is to reveal object composition. Composition is reflected through the spatial composition of display subcomponents and through graphics connecting the subcomponents. In Figure 2.3, the arrows indicate the choices that have been made and reflect the relationships between the schedule and the predefined meals. If those relationships are changed by some other means (e.g., through the schedule display or by some database operation), the arrows change accordingly.

Compositional changes can sometimes trigger notifiers or effects on the set of objects currently displayed, which entail complex display responses, such as those described in Section 2.1.2. For example, suppose there is a restriction that a Meal should not be chosen for more than two DayPlans. The display might convey this constraint by graying out a meal after it has been chosen a second time. This feedback does not directly reflect a relationship, but occurs in response to the creation of a relationship. As with dynamic representation, any feedback on the changes to object composition provides information relevant to the user's task, thus reducing semantic distance.

2.3 Comparison to Existing Systems

Sections 1.1 and 1.2 described areas of work related to supporting displays of complex database objects. This section discusses particular user-interface construction tools that fall in those areas, focusing on their choices for the design aspects in Section 2.1. None of the systems combines all three of the desired design features. In general, the existing tools are not based on a behavioral database model, and the data model used is not easily extended to provide the kinds of semantic information available in behavioral models. As a result, the tools are limited in their ability to describe display responses that reflect changes in database objects, particularly changes in object composition.

Form Generators

Form generators such as FADS [Rowe82] work with record-based data models and thus receive limited information on the structure and behavior of objects being displayed. Since object semantics is mostly imposed by the application, the runtime display support from a form generator is simply to reflect changes to the attribute values within records. Form generators support the design of displays for a particular kind of object, allowing the designer to choose interactors for entering data and to arrange them within a form. They do not provide a way to specify that format should change when certain conditions arise, so a form's format remains static throughout its execution. Thus, format changes are not used to reflect changes in object composition. Rather, the form reflects changes in attribute values, and the user must interpret those values to visualize the object composition.

Seeheim-based UIMSs

The Seeheim UIMSs [Kasik82, Buxton, Olsen83, Olsen86, Green85b, Schulert85, Jacob86, Myers86] have difficulties displaying application objects with complex structure. Typically, a change in one subpart of a complex object will require that its entire display be redrawn. Because of this overhead, the application typically will manage the semantic feedback in object displays instead of having the UIMS do it. The underlying reason for the difficulty is that although the UIMSs can specify compositions of display components, the display definitions cannot express the mapping of display subcomponents onto object subcomponents.

Data-Oriented UIMSs

HIGGENS [Hudson88] and Apogee [Henry88] are data-oriented UIMSs that describe object behavior by modeling objects as active data. Objects are modeled as attribute graphs, where nodes represent data entities and graph connections represent relationships. The type definitions for nodes include attribute-evaluation rules that state how attributes are dependent on those in related nodes. In the UIMSs, a view is a special object that can be attached to a node, and the view drives the construction and updates of the data node's display. This database model does not support the expression of dependencies among relationships, meaning that an update to a relationship link generally cannot be used as a trigger for other actions or as an action triggered by other updates. Thus such behavior must be implemented in the application. Because of the lack of expressiveness in the data model, views do not receive complete information about how the underlying objects are changing and therefore cannot always produce appropriate changes in the display to reflect the objects' behavior. For example, in the display for choosing meals (Figure 2.3), drawing the arrows reflects the relationships between the schedule and the meals, and thus requires information on those relationships.

Like HIGGENS, several other systems are based on a *data-dependency approach*, where much of the activity in an interactive graphics application is described in terms of dependencies among the state of graphical input devices, graphical output, and application semantics. In such systems, a dependency can be viewed as the combination of an *condition* and an *action* to be performed when the condition is true. The systems differ in how the three sets of information are managed and the kinds of conditions and actions that are supported.

- Garrett and Foley developed a system for graphics programming using dependency declarations [Garrett82]. In this system, a graphics application is described as dependencies among graphical input data, application data, and graphical output data. All the data is placed in a relational DBMS that maintains the dependencies when the application executes. The system allowed for conditions that involve the state of specified relations, such as whether tuples have been added, deleted, or replaced in the relations, and whether certain attribute values are equal. Actions include database modifications and calls to external procedures.

- The Serpent UIMS [Bass90] also places the information in a record-based database. Its conditions and actions deal with attributes that define the presentation of interaction techniques. In addition, an application record can be linked to a *view-controller*, which is a logical collection of interactive components such as buttons or text fields. The state of the application data can trigger changes to the view-controller's attribute values or can cause a view-contoller to be created or destroyed.

- The Process Visualization System [Foley86] allows icons to be constructed and logically connected to data values generated from a monitored process and kept in a database. In this system, the conditions involve the state of process variables, and the actions update the icons' attributes, which include colors, fonts, size, position, and visibility.

- The Garnet user-interface development environment [Myers90] includes a toolkit and an interface builder. In Garnet, the three sets of information are defined using an object-oriented programming system that is based on prototypes[1] rather than object classes. The programming system supports the definition and maintenance of one-way constraints, which are similar to the attribute evaluation rules in HIGGENS. Thus, conditions and actions are implicit; whenever a value taking part in a constraint is updated, the underlying system acts to resatisfy the constraint.

- The CONSTRAINT system [VanderZanden89] provides a paradigm called constraint grammars for defining both object connections and dependencies among object attributes. Constraint grammars extend attribute grammars to allow the derivation of attribute graphs (rather than attribute trees) and the definition of multidirectional constraints among attributes. An underlying concept in constraint grammars is that the data structures of the application and the display should be integrated in order to divide the development of graphical aspects from the non-graphical concerns of an application. Thus a constraint grammar defines object composition and display format simultaneously. CONSTRAINT also incorporates an editing model for defining transformations on the graphs that represent the object connections.

To support dynamic representation, it is necessary to express both conditions about the connections between application objects and actions that describe the rearrangement or replacement of subcomponents in a display. The data-dependency systems are mostly concerned with how display attributes are dependent on basic values, not how display format depends on object composition. The condition-action pairs in these systems often constrain basic values in application and graphical objects, or maintain a link between an application object and graphical object. Thus changes in display format generally must be specified procedurally as part of the application program.

The transformations used in CONSTRAINT do support conditions about object connections and actions that can change display format, but for the special case where object composition and display format correspond exactly. A transformation consists of a selection pattern of graph nodes representing a group of connected objects and a set of replacement rules expressing how to modify the subgraphs that match the selection pattern. A drawback of integrating application and display objects is that it may become difficult to support multiple display representations

[1]See Lieberman's paper [Lieberman86] for background on the concept of prototypes.

for an object type. The object semantics must be re-specified for each possible representation, and consistency among multiple definitions must be maintained.

The Filter Browser [Ege87] is another data-dependency system for defining object displays. A *filter type* is a package of constraints between a *source* object and *view* object, along with a type specification for both objects. A filter type can declare subfilters, thus the type system includes filter constructors (sequence, iteration, and condition) that can define changes in the number of subfilters present at runtime. Certain kinds of format changes can be described using filter constructors, and there is no requirement that the source and view object have identical structure. However, many of the basic display behaviors are encapsulated in *filter atom* types that are provided by the system, thus those behaviors are not definable by the display designer.

OODB Browser Generators

Most display generation tools that produce object browsers lack support for dynamic representation as well. The LOOKS system [Plateau89] is part of the programming environment for the O2 [Deux91] OODB system. LOOKS provides three types of predefined generic views to display an object: icon, level one, and all-you-can-show. A new form or *look* for an object class is created using a customization editor to build the display graphics, which consist of a background and a set of slots. The programmer also defines the mapping between the object components and the slots in the form, and thus can choose which components to display and their arrangement in the form. However, a display instantiated from a look cannot change its format during execution. Such a change must be made by the application, and requires instantiating another look having the new format and substituting the new look for the old. The KIWI OODB system [Laenens89] provides a customization package of predefined frame and window classes that include buttons, text editors, tables, and forms that serve as containers for other windows. Customized browsers are created by composing windows hierarchically using KIWI's database programming language. As with LOOKS, one cannot specify that a browser should change its layout format during execution.

FaceKit [King89] is a user-interface toolkit built on top of the Cactis OODBMS [HudKing87]. FaceKit includes a menu-driven tool for editing a *representation definition* that describes the appearance for a database object or schema. The display designer can create a new appearance or use a representation provided by FaceKit. For example, default representations exist for displaying a schema as a graph diagram or as nested forms. Another tool supports *operation* *definition* for the user interface, in which actions are bound to menu items or sequences of user input. The actions invoke methods or queries, written in the Cactis DDL or in C, that construct the objects' displays and provide any interactive behavior. FaceKit supplies some low-level constructs (e.g., opening a window or printing strings) for use in defining the actions; like the Seeheim UIMSs, FaceKit does not provide high-level abstractions for managing complex-object displays.

The Smalltalk Interaction Generator (SIG) [Maier86] focuses on the ability to adjust display format to match changes in object composition. In SIG, a *display type* is a declarative description for browser-like displays created for instances of a particular class. A display type consists of one or more *recipes* that describe a different format for the display. Each recipe includes a selection condition representing a certain state of the displayed object thus determining when during display execution the recipe should be used. An *abstract view* is a Smalltalk object that implements a display; it monitors the displayed object to detect selection conditions listed in a display type and reconfigures itself accordingly. Although SIG supports description of format changes, it lacks support for describing behavior within the display components. A recipe holds a list of ingredients, each of which defines some type of view. SIG supports customization for particular types of views such as text, list, and form views. However, most of the behavior of these views is fixed. Custom views may be defined, but are created by programming in Smalltalk.

Toolkits and Application Frameworks

User-interface toolkits (e.g., the Macintosh Toolbox or the X toolkit) provide predefined interactors to use as building blocks for a display, but do not give much support for conceptualizing or constructing an entire display. Some object-oriented toolkits, such as InterViews [Linton89] and the GRaphical Object Workbench (GROW) [Barth86], provide abstractions for describing the visual arrangement of interactors, but not how their behaviors compose. Toolkits are considered difficult to use because they contain a large number of building blocks (in the form of procedures or classes) and often it is unclear how each should be used.

An *application framework* is a class library designed to reduce the difficulty of using a toolkit by providing classes whose instances act as applications. The application object provides the basic functionality that is required or common for applications in a particular environment. An application object utilizes toolkit building blocks to perform its activities, thus removing the programmer's need to know about certain building blocks. In some cases, the functionality of certain building blocks is made more apparent because the programmer has an example of how they fit into a working application.

For example, MacApp [Schmucker86] is a framework that enforces standards for Macintosh applications. The main classes for building an application are *TApplication*, *TDocument*, subclasses of *TView*, and *TCommand*. A TApplication object supplies the functions of opening, initializing, and closing an application. In addition, it holds together all the framework objects that make up the application. A TDocument object provides file-related functions such as opening and saving. Various kinds of TView objects manage an application's windows and their contents. For example, a TWindow handles window resizing, placement, opening and closing. Other kinds of TViews are TControls, which represent interactors such as scrollbars, text editors, popup menus, and buttons. A TCommand object provides the general mechanics for invoking a command or requesting an undo or redo of the command. The programmer supplies the actions to be performed when a request is received.

The VisualWorks development environment [ParcPlace] includes an application framework for creating Smalltalk applications with GUI features. The framework provides classes for visually arranging interactors, however the display designer does not need to use them directly, since the environment includes a graphical editor through which interactors can be positioned and assigned certain visual and behavioral properties. The ET++ application framework [Weinand89] is also part of a programming environment. It provides some classes for composing the behavior of interactors. For example, one ET++ class supplies the ability to forward input events to one of its components depending on the event type. Subclasses of this class control communications between the components in predefined ways.

Application frameworks have good potential to support complex display responses and dynamic representation, since a framework can continally evolve and provide more abstractions through the addition of new classes. However, the design of application frameworks is generally not aimed at separating the user interface from the rest of the application, and thus they represent an approach to user-interface support that is fundamentally different from the goals pursued in ODDS.

Visual Construction Environments

Another type of user-interface tool is an environment for constructing displays by composing the layout and behavior of predefined components using direct manipulation techniques instead of a specification language. Two examples are the NeXT Interface Builder [Webster89] and the Fabrik visual programming environment [Ingalls88], which both allow a certain amount of display behavior to be expressed graphically. Fabrik uses dataflow diagrams for specifying the communication paths among user interface components and computational components, thus showing how the components work together as a unit. In the Interface Builder, the designer executes a sequence of keyboard and mouse inputs to establish a *target-action connection*, which specifies that manipulating one component will cause a message to be sent to another display component or application object.

These paradigms for describing behavior do not extend gracefully to support complex display responses, particularly changes in display format. The paradigms focus on describing actions that display components perform on input data or application data, but do not address actions that manipulate the display components themselves. In Fabrik, one might describe a format change by defining a special computational component that creates, deletes, or manipulates components as necessary. Thus, the format changes are described programatically, not via the dataflow paradigm. Similarly, in the Interface Builder, a format change must be defined by writing code that manipulates the components of the display. Since the display designer constructs an image for the display, then describes behavior based on that image, actions that change the format of the image may invalidate behaviors that were described visually. Such conditions must also be dealt with programmatically.

The GemStone Object Development Environment (GeODE) includes a Visual Program Designer for customizing the behavior of forms that make up an OODBMS application. The Forms Designer provides a variety of interactors that can be placed on the form and allows for the definition of certain characteristics of the interactor's appearance and behavior. Visual programs represent operations over fields, forms, or data. A dataflow model indicates the sources for the operations and where their results are placed. Some types of format change can be described since the Visual Program Designer provides building blocks that can manipulate the visibility or positioning of fields.

Chapter Summary

This chapter established basic requirements for ODDS design. A major goal behind the requirements is to design a user-interface tool through which a display designer can describe various kinds of semantic feedback. Semantic feedback plays a key role in giving displays a feeling of directness, as shown in Section 2.2. In particular, feedback concerning object composition is important for displaying complex entities. Without the proper presentation of composition, the entities will appear as disjoint pieces of data, and the user must keep track of the composition in his or her head.

Section 2.3 described existing user-interface tools, and showed that they provide limited support for semantic feedback, concentrating mainly on feedback of attribute values. Basic requirements for ODDS include supporting feedback that involves coordination among interactors and changes in display format, as these types of feedback are useful for presenting object composition. To describe and execute these types of feedback, ODDS needs access to the displayed objects and information on their semantics. Thus, another of ODDS' basic requirements is to manage the displayed objects in a behavioral DBMS.

Chapter 3

Objectives and Overview of ODDS' Design

This chapter presents the objectives and issues underlying ODDS' design, then presents the basic concepts of the system. Sections 3.1 and 3.2 discuss the objectives, relating them to the three design choices presented in Section 2.1. These design choices are key to resolving the conflict between supporting semantic feedback and maintaining modularity between display descriptions and applications. To recap, the choices are to use a behavioral database model for defining object semantics, to capture complex display responses within the display descriptions, and to support dynamic representation, i.e., capture how a display's format may change during execution.

The objectives are discussed in terms of the two main functions that user-interface construction tools generally fulfill. The first is to provide techniques for describing the desired capabilities of displays. The second is to construct and execute displays based on the descriptions created by the display designer.

Section 3.3 provides introductory information on the display descriptions and the runtime architecture in ODDS, setting the context for following chapters.

3.1 Design Objectives for the Specification Framework

The first group of objectives relates to mechanisms for describing displays. The task of designing the specification framework consists of choosing a set of constructs to be used for building up a display description.

3.1.1 Expressiveness for Complex Display Responses

The specification framework must include constructs for describing how the display will reflect the semantics of underlying database objects. This requirement stems from the basic design choices to capture complex display responses and to support dynamic representation.

A closer look at feedback is needed before discussing the specification framework's expressiveness. A user's inputs are ultimately interpreted as a request to perform some action concerning the displayed objects, either to change them or obtain more information about them. *Lexical and syntactic feedback* [Foley82] provide information about how input sequences are being parsed by the user interface. For example, a menu entry may be highlighted when the user clicks on it, signalling that the user interface received the mouse click and that a selection was made in the menu. As another example, the interface may produce messages saying that an input expression does not make sense, or that the interface needs more input to understand what action to perform.

Once the user interface has detected and serviced a request for an action, *semantic feedback* shows the effects of the action by reflecting changes made in the underlying objects. An object's display image can be considered an *external representation* for that object [Anderson86, Ege87]. With this viewpoint, producing semantic feedback is seen as an operation that translates the updated object to its external representation, thus presenting the object's new state. Consequently, a description of a display's semantic feedback defines what takes place in these translations.

A specification framework's expressiveness with respect to semantic feedback can be discussed in terms of the kinds of translations that may be described through the framework. A basic requirement for defining any kind of translation is a way to identify a certain position in the displayed object to be a source of data values for the external representation. Such a position can be described by a path relative to the object, consisting of a sequence of unary messages. A path is denoted by an expression #(message1 message2 ... messageN), and is relative to a given *root object*. The first message in the sequence comes from the root object's protocol, the second message is from the protocol of the first message's return value, and so on. A *path value* is the return value of the final message. For example, the path #(currentNutrition fat) relative to a Diet object is illustrated in Figure 3.1, and the path value is 37.5.

The simplest kind of translation is one where a change in a path value affects its corresponding representation in the display. For example, when the amount of protein for a meal is changed, the number shown in the display also changes to show the new amount. Another kind of translation is one that requires calculating a new value from values in the object, such as an average or sum, and placing a representation for the new value somewhere in the display.

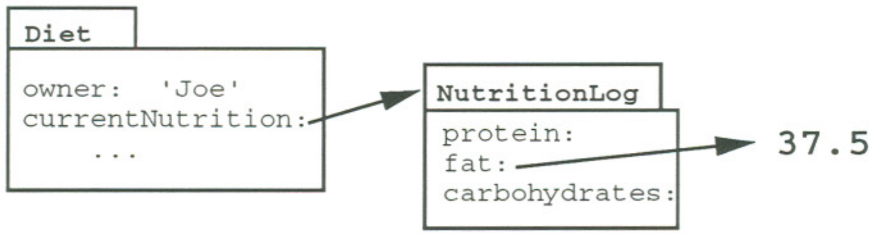


Figure 3.1: Example of Path

A more complicated kind of translation involves updating display attributes, such as color or spacing, based on an updated path value. For example, in the display for nutrition status (Figure 2.2), the fill pattern for a value bar depends on the amount of the nutrient it represents and how the amount relates to the upper and lower limits for that nutrient. A description of this translation involves several details including how to evaluate the relation between the amount and the limits and how to associate the result with the proper fill pattern.

Translation can also involve the coordination of multiple display changes to reflect a particular change in the object. In the nutrition display, changing the amount of a nutrient can cause two separate display responses: adjusting the height of the value bar and changing its fill pattern.

Finally, there are translations based on the relationships between objects, i.e., translating the relationships into some visual representation. For example, an arrow in the display of meal choices (Figure 2.3) is a graphic representation of a relationship. A relationship between two objects might also be represented by positioning their display images in a certain way; e.g. positioning a meal display within the display of a DayPlan object indicates that the meal is the breakfast, lunch, or dinner for the DayPlan (Figure 2.1). When a relationship is represented by relative position, performing the translation can cause a change in display format.

In summary, to support the range of translations described above, the specification framework must include constructs to:

- make references to subparts of the displayed object
- perform computations using values from the object as arguments, and use the result of a computation to affect aspects of the display image or its future behavior
- coordinate multiple display changes that occur in response to a single change in the object

- specify options for the spatial composition of the display subcomponents that can result in adding, removing, or replacing subcomponents, and
- specify when to change from one format option to another.

3.1.2 Declarative specification

A major issue in choosing the specification constructs for the framework is finding a balance between keeping the specification techniques at a high, abstract level and giving the designer sufficient expressive power. High-level, declarative specification techniques are desirable for modifying the display features expressed in a description. Modifications are less complex when one can understand and describe a display in terms of what its activities are rather than how they are implemented. A high-level perspective of displays can also make opportunities for reuse more apparent. Another benefit of declarative specification is that usage of the descriptions does not require that an application be implemented in a particular language; this independence is important since DBMSs typically have interfaces to several languages. With a toolkit or application framework, a particular programming language is used to define displays, thus the application program using the tool must also be written in that language (or be able to interface with procedures and data types of that language).

It has been noted, however, that tools using declarative specification techniques are usually limited somehow in what they can express [Myers89]. For example, a tool may supply a fixed set of interactors as building blocks whose appearance and behavior cannot be changed in any way. The designer should have control over such details so that the display can be tailored to the semantics of the object being displayed.

In designing displays of complex objects, a substantial amount of expressive power is needed because of display requirements related to object relationships. Some requirements that have been identified [Maier86] include: the ability to adjust display format to match object composition; the ability to accommodate displays with arbitrary levels of structure, having no bound on depth of nesting; the ability to maintain consistency between multiple displays of an object (which typically arise because of multiple connections to an object). To meet such requirements, the designer must be able to group several interactors into a single component and must have control over how components are related in terms of visual placement and behavior. In current tools, specification languages that offer the necessary level of control tend to resemble imperative programming languages [Hudson88, Sibert86].

Several tools [Ingalls88, Webster89, Myers90, ParcPlace] include a graphical editor that provides a high-level approach to specifying display appearance. The difficulty of balancing declarative specification and designer control lies mainly in defining constructs for interactive behavior. Two levels of behavior should be addressed in defining constructs for the specification framework: the behavior of interactors and the behavior to coordinate interactors.

The task of defining the constructs is basically to categorize the activities that occur in display execution, then generalize each type of activity by distinguishing which of its characteristics should be definable by the display designer and which parts remain fixed across activities of that type. Ideally, the specification construct would hide much of the programming detail involved in implementing the activity, and the definable characteristics would allow the designer to describe many variations of the activity to accommodate different situations.

3.1.3 Modularity for Supporting Incremental Development

Another factor that helps simplify modifications to display descriptions is the ability to define different types of display features semi-independently. In particular, a certain amount of independence exists between the two levels of behavior; some parts of an interactor's behavior can be changed without having to affect the behavior that coordinates the interactor with others. The specification framework should reflect this independence by separating the descriptions for the two behaviors. A similar independence holds between the visual features and the behavior of an interactor. An interactor behavior can be attached to different images; e.g., scrollbars may have different looks but operate in exactly the same way.

3.1.4 Support for Reusing Descriptions

Other objectives addressed in designing the specification framework relate to how the descriptions can be reused. As previously stated, one of the basic goals of ODDS is keep a separation between the display descriptions and the application code, since this separation promotes reuse between applications. Reuse should also be possible when creating a display description. The framework should include constructs that enable a description to designate another description as the definition for a subpart in a display. A different kind of reuse to support is being able to copy some or all of a description, and make modifications to suit one's current need. The difficulty level of performing this reuse is dependent on the ease of identifying the description or subpart that is similar to one's needs.

Another objective is to support reuse of descriptions in the event of database schema changes, which may be changes to a class definition or the addition of subclasses. It is desirable that descriptions for a newly defined class be able to use an existing description or some of its parts. In particular, the existing descriptions for a modified class should still be usable with modifications commensurate with the changes in the class. Similarly, descriptions for a new subclass could make use of descriptions for its superclass. A related issue to consider when new classes are added to the schema is that display descriptions of other classes may be affected by the change. A dependency can exist if some other class refers to the modified class (or one that is subclassed) and one of its displays includes a subdisplay for the modified class. Ideally, modifying a class would not require changes in the display descriptions for other classes.

3.2 Objectives for the Runtime Architecture

In addition to a specification framework for describing displays, ODDS includes runtime support for constructing and updating a display based on its description. As discussed in Chapter 1, the basic objective is to design an architecture where both the display system and the application have immediate access to information needed to perform their functions. Having immediate access means that the two components do not need to ask each other for the information they require. To exhibit behavior that contributes to directness, the displays require access to the displayed data's state *and* information about its semantics, such as data dependencies and type information. Therefore, as explained in Chapter 2, a behavioral database model is chosen for representing the central data.

ODDS' runtime architecture defines the main functions performed by the three major components (i.e., display system, central data store, and application) and the interactions among those functions. Thus, it defines what runtime services are provided by ODDS, and the means by which an application invokes the services. It also defines how ODDS obtains information or services needed from an application when executing displays. In addtion, it defines how functions within ODDS interact with the central data store; i.e., what those functions require of the data store.

An objective in designing the runtime architecture is to identify the system functions with sufficient generality, making them applicable to various environments in which a display system could be used. Thus, the defined functions should not be tied to the requirements of a particular

OODBMS architecture or window platform that might be used to implement the system.

The remainder of this section discusses objectives regarding the functionality in the architecture. Three main features motivate the presence of specific functions in the architecture. These features are: the shared data arrangement, the use of a behavioral database model for the central data store, and the execution of semantic feedback as defined in a display description. The functions needed to support each feature are discussed in the following subsections.

3.2.1 Shared-Data Architecture

One essential task in the runtime system is to record information about changes made to the central data store. In a shared-data architecture, the display system and application do not have sole control over the objects that they work on. Since a change in an object may require a responsive action by either the display system or application, each must be able to determine what objects were affected by the other. Having the components look for such changes would result in a large overhead, therefore the architecture should include a mechanism to record changes when they occur and report the changes to the display system and application.

While the shared-data arrangement addresses the need for the application and displays to communicate about the state of displayed objects, there are other reasons for the two parties to make requests of one another. To use displays described in ODDS, an application must be able to request the generation of particular displays and coordinate its activities with certain coarse-grained display operations such as activation and deactivation of the displays. Thus another required function in the architecture is to facilitate the coordination between the two parties. This function includes providing each party with information about the changes to the displayed objects made by the other party.

The functional requirements of an application using ODDS are also affected by the shared-data arrangement. The application must be written with the consideration that its database objects may be updated by an external agent. In addition, the architecture should include some means for the application to use ODDS' services for displaying application objects that are not stored persistently in the database.

3.2.2 Using a Behavioral Model for Shared Data

For objects in a behavioral database model, the task of detecting state changes is complicated by side effects that may occur in a method's execution. When a display or application sends a message to a database object, the executed method can send messages to the object's attribute values, connected objects, or any message arguments. Those messages in turn may affect yet other objects. Therefore, the update-reporting mechanism must be able to detect changes from side effects as well as direct requests from the user or application.

A task related to udpate detection is determining which state changes are relevant to the current displays and thus are needed by the display system. The relevant changes are not limited to those in currently displayed objects; changes in non-displayed paths can also affect the path values presented in a display. Two situations make a display dependent on non-displayed paths. One is when any message in a displayed path returns a value that is derived from the values in one or more non-displayed paths. Another situation is when any prefix message in a path (i.e., any message but the last) returns a value that is not displayed. As shown in Figure ?, changing an intermediate value along the path can affect the final path value.

A decision to consider is whether the filtering of relevant information should take place as part of update detection or after all state changes have been detected. The advantage of filtering the set of objects monitored for changes is that the effort and time spent for update detection can be reduced. In this approach, the update-detection needs information on which objects and paths are relevant, and must keep the information current as changes occur while the display is running. An alternative approach of choosing relevant information from all gathered information (a list of updated objects and affected properties) requires that the display system check the paths relevant to the current display, and must therefore know how the updated objects are connected. The decision requires examination of the tradeoffs with respect to the required maintenance and volume of information transferred.

3.2.3 Runtime Support for Semantic Feedback

The implementation of semantic feedback in an ODDS-generated display consists mainly of translating changes in the displayed database objects to the appropriate display changes. To perform these translations, the runtime system must maintain information on the mapping from a database object to the runtime object(s) responsible for displaying it. In particular, format changes require that the runtime system be able to identify the display images of database objects involved when some object connection is updated. The runtime system needs this information to update the visual relationship among those images, thus reflecting the updated connection.

Maintenance of the mapping information also includes keeping track of what source objects are currently being displayed. The set of displayed objects might change due to updating an object connection or to a user request. Tracking this set is necessary for supporting the detection of source changes relevant to the displays, as discussed in the previous subsection.

3.3 Design Overview

3.3.1 Form of Display Descriptions

This section discusses design decisions concerning the form of the display descriptions. As stated in Section 3.1.2, ODDS seeks to provide declarative specification, allowing the display designer to work with abstractions that match the design task more closely than an imperative programming language. Declarative display descriptions could take various forms, for example, as code written in a declarative language or as executable displays produced through a visual construction environment.

In ODDS, display descriptions are in the form of complex objects that are interpreted by a runtime system to generate and execute displays. As a complex object, a description is similar to a parse tree that represents the meaning behind source code as a composite structure of nodes having various types. Such a description is a step easier to interpret than a language-based description, since it does not need to be parsed. Using the form of complex objects also has the advantage that descriptions are easily converted in some other form such as text or a display image.

A display constructed through a visual environment is its own description and does not require a generation step before execution. However, with this form, certain aspects of the display description are not easily manipulated, as described in Section 2.3.6. For a description in the form of complex objects, meaning is encoded in object connections and in the types of the connected objects. Therefore, a wide range of display semantics can be modified by changing connections and values within the description objects.

Display descriptions are stored in the database with the displayed objects, making them accessible to both the database applications and the runtime system. The runtime system needs access to the descriptions to handle a display's format changes incrementally. Otherwise, the runtime system would have to obtain and store information on all possible formats even though some formats are never used in that particular instantiation of the display. Alternatively, if the descriptions are not kept in the database, the runtime system might maintain them in files where they would also be accessible to the applications. However, the runtime system would have to duplicate data modelling and management capabilities already present in the DBMS.

Other systems (e.g., FaceKit [King89]) have stored display descriptions in a database in the form of methods that draw the displays and handle interactive behavior. Such descriptions are procedural, thus this approach is not adopted for the reasons given above. In addition, database methods that implement display functionality do not provide the desired separation between display definition and object semantics. Having descriptions as complex objects makes them distinct from the methods that define the semantics of the displayed objects. In ODDS, the definition of how and when display execution invokes database methods is expressed through certain types of description objects, rather than simply being another message invocation embedded in a display method.

A database object being displayed, by an ODDS-generated display is called the display's *source object*. An ODDS description object describes displays for source objects of a particular class, called the descriptionUs *source class*.[1] Since a display description is defined at the class level, it is a template or partial description for a display's appearance and behavior. It defines the display features that are common to instances of the source class, but does not hold the data values from a particular instance. Thus the descriptions are called *Outlines*. An Outline consists of two kinds of information: 1) a description of the display's graphical image and 2) a behavioral description; i.e., the display's reactions to significant events such as user input or changes in source objects. Accordingly, there are two hierarchies of description classes, the `Layout` and `Interaction` classes, whose instances are part of Outlines. To distinguish these description objects from source objects, they are called Layout and Interaction *specs*.

Several Outlines can be assigned to a class, so its instances can be displayed in different ways, depending on the context of the display. Multiple perspectives are often needed for objects with complex semantics, because a single representation generally will not be appropriate for all possible operations on an object.

Several advantages are gained by making display descriptions part of the database [Anderson86]. Associating display descriptions with classes provides a more comprehensive information base on the source objects. The descriptions become part of the semantic information describing the objects. Secondly, descriptions can have a direct pointer to semantic information stored

[1] A display description could also be used for instances of the subclasses of its source class.

in database classes. Changes to the referenced information will affect the descriptions automatically. In effect, some semantic information is being incorporated into the display specification, which works against the reusability because the references to a specific class must be checked when reusing the specification (or some part of it) for a different source class. However, because the descriptions are in the database, it is possible to use data management facilities to keep track of where the descriptions are dependent on information specific to a certain source class.

Finally, the descriptions can be examined, modified, and displayed just as other objects are. An interactive editor for creating new descriptions could be built as a database application that uses the display descriptions associated with the Outline, Layout and Interaction classes. In addition, the specification objects will be available to other database tools that assist with application programming.

3.3.2 System Architecture for Runtime Support

This section provides an overview of ODDS' runtime support for creating and managing displays on behalf of a database application. The ovals in Figure 3.2 represent the major components that make up ODDS' runtime support system, hereafter called ODDS-Runtime. The database acts as a central resource and a means of communication between display execution and the application program. Since both sides have read and write access to the database objects being displayed, the modifications made by one side are visible to the other side. The application might act based on changes made through the displays, and the displays can act based on changes made by the application. As a result, the two sides can communicate through modifications to database objects.

The *Application Communication Layer* is a programming interface through which an application invokes the services of both ODDS-Runtime and the DBMS, providing the appearance that the application interfaces to a single system. In practice, different versions of the Application Communication Layer could be developed for different programming languages. Thus, ODDS could be used by applications written in any of those languages. The display services provided by ODDS are described in Chapter 5.

The functions of the *Control Manager* include establishing a connection with a client application and forwarding requests for display services to the appropriate parts of ODDS-Runtime. Thus, the Control Manager coordinates the exchange of control between the displays and the application. Another role of the Control Manager is to insulate the other system components

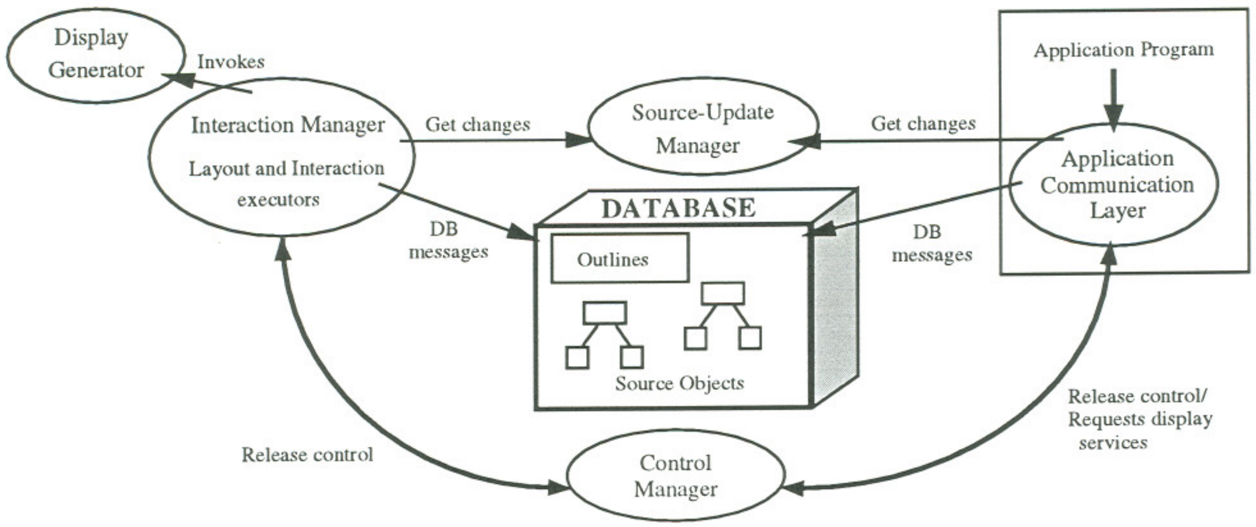


Figure 3.2: Components of the ODDS-Runtime System Architecture

from differences in OODBMS architecture. In the ODDS prototype, the application runs as a separate process from the database; however other possible configurations exist for OODBMSs in general. The application, database session, and display system may be all in one process. The latter case occurs if applications are written in the database language instead of an external language, or if the OODBMS provides the ability to link an application with a session. In that case, interprocess communication would be replaced by local mechanisms.

The *Source-Update Manager* monitors the source objects currently being displayed, and provides update information for either the application or to the Interaction Manager when appropriate.

The *Display Generator* is invoked when the application or user[2] requests that a database object be displayed, and also when the display format is changed after the display is created. The *Interaction Manager* is responsible for updating the displays as described in Outlines. These two components encapsulate the functions that deal with graphical input and output. Like the Application Communication Layer, specialized versions of these components could be implemented, enabling ODDS to support displays that comply with a specific GUI standard (such as Macintosh or Microsoft Windows).

When an application requests that a display be created, it supplies the name of an Outline and the source object to be displayed. The supplied information is used by the Display

[2]A generated display could provide the ability to spawn other displays, thus a user action may indirectly invoke the Display Generator.

Generator, which generates a set of runtime objects, or *executors*, that draw the screen images and carry out the behaviors as defined in the Outline. To construct executors, data from the source object is merged with the Outline's partial description, producing a description of the source object's display. The structure and content of the generated executors essentially parallels that of the specs in the Outline. Accordingly, the classes for display executors, called the `LayoutExec` and `InteractionExec classes`, have an exact correspondence with the `Layout` and `Interaction` classes for display specs. The display generation process is described in detail in Section 7.2.

Since the Outline specs and the executors generated from them are similar in many ways, I emphasize the distinctions between them to help clarify discussion in subsequent chapters. One distinction is that an executor represents a single display instance, while an Outline is a template for many display instances. Another way of seeing this distinction is that an executor is associated with a particular source object, while an Outline is associated with the source class. A second distinction is that executors have values that change as a display executes, while an Outline is static during display execution. An Interaction spec in an Outline refers to a Layout spec and describes actions of the display in terms of updates to the Layout specs. In actuality, the updates described in the Interaction specs are not applied to the Layout specs, but to the executors generated from those specs. A third distinction is that Outlines reside in the database while executors are objects within the ODDS-Runtime. In short, Outlines are persistent and do not change as a result of display execution, while executors are transient objects that exist only during display execution and are updatable during that time.

During display execution, the Layout and Interaction executors function as part of the Interaction Manager. An executor interprets the information copied from its spec counterpart, carrying out the semantics expressed by the spec. Other tasks performed by the Interaction Manager include interfacing with input devices and with the database to produce events that will activate an appropriate Interaction executor. The Interaction Manager also invokes the Display Generator when the format of an existing display is to be altered.

The architecture for ODDS-Runtime follows the general pattern of a data-oriented UIMS (see Figure 1.2). A separation exists between the user interface and the application, but it is one that provides sufficient information about the displayed objects to allow the ODDS-Runtime to produce many types of semantic feedback without having to consult the application. The separation also provides the advantage that ODDS-Runtime could service applications written in various programming languages and could connect to application processes in various locations.

The remaining chapters are organized as follows. Chapter 4 describes the spec classes that make up the ODDS specification framework In addition, the underlying model used for building a display description from instances of those classes is described. Chapter 5 describes usage of ODDS from the perspectives of display design and application design, discussing the degree of independence achieved between the two tasks. The detailed functionality and implementation of the system components are presented in Chapters 6 and 7. Chapter 8 describes the displays that were constructed using the ODDS prototype, and presents some evaluation of the system with respect to its expressiveness, usage, and runtime performance.

Chapter 4

Expressive Capabilities of Outlines

The classes making up the ODDS specification framework are based on a particular model for the construction of interactive displays. The description of a display's functionality encompasses many features, thus it is important to have a conceptual organization that identifies the kinds of features to be specified and how they relate to one another. Such an organization provides guidelines for integrating subparts into a complete display description; thus it promotes a development process where the designer can break the work down into smaller subtasks, and possibly use predefined pieces for parts of the display description.

The model also provides help for understanding an existing display description, which is important when modifying or reusing the description. A description might be modified to correct an error or to change some feature of the generated display. A good understanding of the display description helps the designer know which part of the description should be changed to achieve the desired effect. Similarly, the appropriate part of a description must be found when one wants to duplicate some feature from another display.

4.1 The ODDS Construction Model

The display construction model sets apart three main areas to address when defining interactive displays: image presentation, action sequencing, and support for source semantics. Each area denotes a certain kind of activity that takes place within a display. Figure 4.1 presents the construction model and shows how specs associated with the three areas are interrelated when constructing a display description. Note that the arrows represent references between the specs comprising a description, and should not be interpreted as data or control flow occurring during display's execution.

The activities denoted by *image presentation* are the graphics operations and algorithms

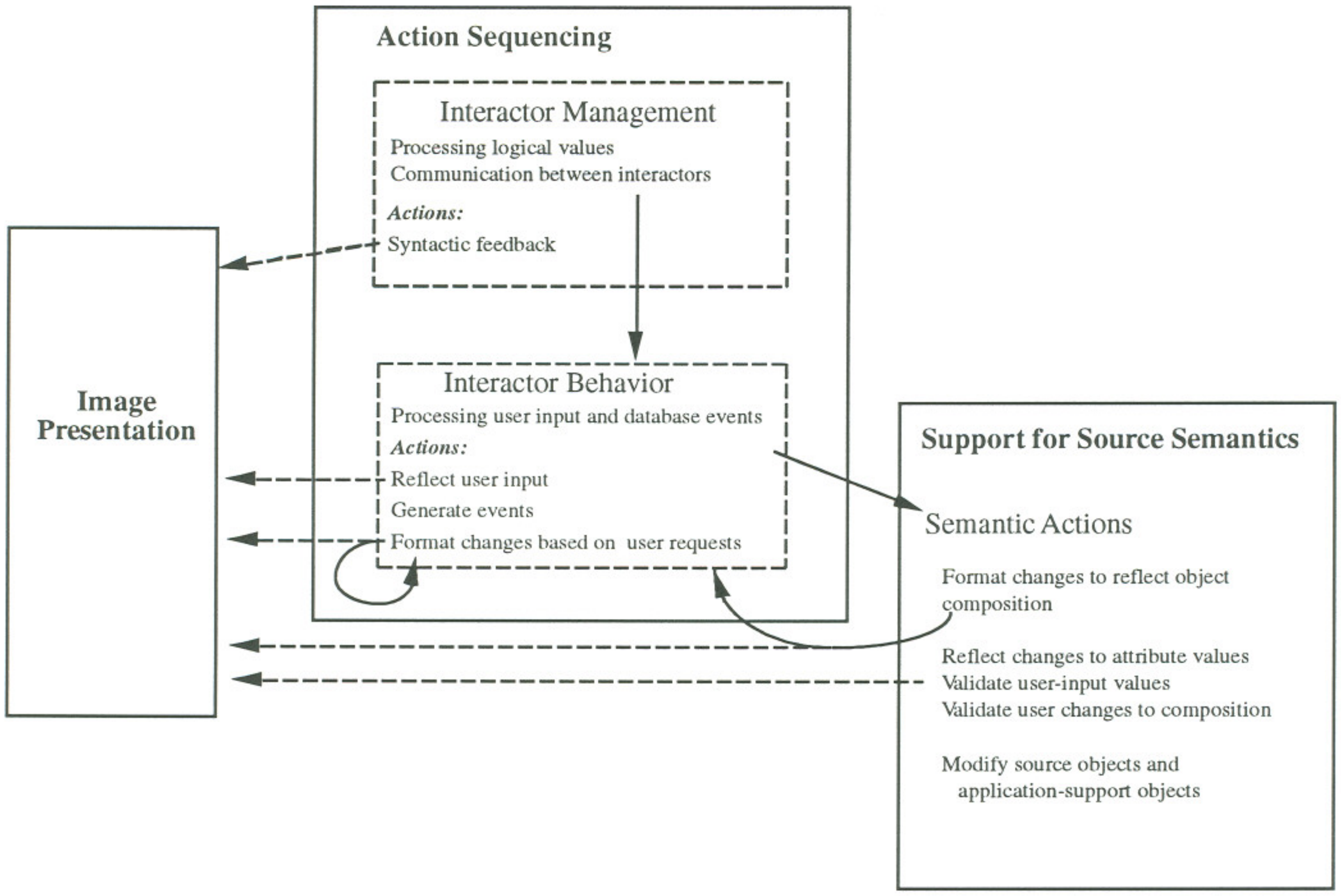

Action Sequencing
Interactor Management
Processing logical values
Communication between interactors
Actions:
Syntactic feedback
Interactor Behavior
Processing user input and database events
Actions:
Reflect user input
Generate events
Format changes based on user requests
Image Presentation
Support for Source Semantics
Semantic Actions
Format changes to reflect object composition
Reflect changes to attribute values
Validate user-input values
Validate user changes to composition
Modify source objects and application-support objects

Figure 4.1: Construction Model for Interactive Displays

that render a screen image. These activities do not include making decisions about what is to be drawn; rather, they are based on drawing requests made by other kinds of activities. The various activities that involve drawing requests are indicated by the dashed arrows in the figure. To express a drawing request, the description (i.e., spec) of the requesting activity refers to a description for the affected image.

The activities in *action sequencing* include processing user-input or database events to determine what actions take place after an event has occurred. Action sequencing also includes the display feedback informing the user that events have been received and are being processed. The activities in *support for source semantics* include both the display's interaction with source objects and the display feedback that reflects the source objects' state or semantics.

The three areas of ODDS' construction model are similar to the components of the Seeheim model. One major exception is that ODDS separates graphical output from input handling, whereas these activities are combined in the Seeheim model's presentation component. ODDS associates input handling with action sequencing, grouping together all activities related to display behavior and distinguishing them from display appearance. This division is intended to promote reuse of a particular behavior with different images, and vice versa. The construction model also supports modularity between semantic and non-semantic activities, which facilitates the reuse of images or behaviors in displays for different source classes.

The specification framework includes classes for describing each of the three areas. The spec classes are designed to have a level of abstraction that makes them independent of several runtime aspects. First, the descriptions of user-input handling and drawing operations are not tied to a particular window environment. Thus, implementation details for input events and graphic images are not considered by the display designer and instead are confined to ODDS-Runtime. Second, connections to source objects are described abstractly, using the notion of paths. The way in which messages are sent to objects (either for obtaining state information or updating the objects) is also not a concern of the display designer.

The following subsections provide more detail on action sequencing and support for source semantics.

4.1.1 Action Sequencing

Action sequencing denotes control over the sequence of actions that occur in the display. In the Seeheim model (see Section 1.3), action sequencing is described as part of the dialogue control component. In that model, the operation of a display is likened to language interpretation: a stream of input is parsed to identify tokens and meaningful sequences of tokens, then actions are executed based on the tokens. The executed actions include asking the presentation component to update the screen and requesting services of the application. Some Seeheim UIMSs [Olsen86, Green85b, Pfaff85] base descriptions of dialogue control on state-transition diagrams or context-free grammars.

Although the language-interpretation approach is suitable for user interfaces that are single-threaded (e.g., a menu-driven interface), the approach can lead to complex and unmanageable descriptions for multi-threaded dialogues, which are common in direct-manipulation interfaces. For this reason, subsequent UIMSs introduced an alternate view of dialogue control, breaking it down into a collection of autonomous dialogues where each dialogue is relatively simple [Jacob86, Sibert86, Binding87, Hill86]. As a result, dialogue control becomes decentralized.

The ODDS construction model adopts the latter approach, distinguishing two levels in which sequencing is managed. The lower level is *interactor behavior*; it includes the handling of input devices and the input processing performed by individual *interactors*, which are interactive components such as menus, buttons, text fields, or scrollbars. (Other common terms for interactor include *logical input device*, *interaction technique*, and *widget*.) An interactor processes the events generated from input devices, then produce an abstracted unit of input, or *logical value*. For example, a slider produces an absolute number or a percentage; a screen button produces a command request. In the course of processing the events, feedback is provided to the user through changes in the interactor's image. At certain points in its operation, an interactor makes a logical value available to other parts of the display for further processing.

Also represented in interactor behavior is the triggering of actions that either cause or respond to changes in database objects. This type of activity is often associated with an interactor that represents a particular part of a source object; e.g., a text field that represents the `name` attribute of a Food object. Such an interactor often has behavior that involves both input processing and databased-related activities. However, an interactorUs behavior might exclude input processing; for instance, an interactor may be "read-onlyUU, meaning that it responds only to database changes and not to user input.

The higher level of action sequencing is *interactor management*, which represents the coordination among several interactor and database-related behaviors. Interactor management involves the communication and the transfer of control between behaviors. The course of action at this level is based on *internal events* sent out by those behaviors. An example of an internal event is when an interactor produces a logical value. The activities in Interactor Management are driven by internal events, and are not concerned with how the internal events were generated.

The actions invoked by action sequencing are divided into *non-semantic actions* and *semantic actions*. Non-semantic actions are display changes that do not affect the underlying source objects. For example, scrolling through a list display does not change any source objects. These types of actions result from a user's request to change the way objects are being presented rather than changing the objects themselves. They include providing lexical or syntactic feedback that informs the user about the state of input processing.

Most actions are invoked at the level of interactor behavior. As mentioned earlier, actions performed by an interactor include reflecting user input and providing logical values to other parts of the display. In addition, an interactor can change its properties, such as its visibility or whether it will respond to user input. Other kinds of non-semantic actions include changing display format when requested by the user and opening related displays. In Figure 4.1, descriptions of format changes refer to both image descriptions and other behavior descriptions since format changes can alter a display's behavior as well as its appearance.

At the level of interactor management, internal events are relayed from one behavior to another. In addition, actions provide syntactic feedback that presents status information about input processing that involves several interactors. For example, suppose that a display requires a particular ordering for entering data into a group of interactors. One way that the the display might provide syntactic feedback is to print an error message if an incorrect ordering is detected.

4.1.2 Support for Source Semantics

Semantic actions make up support for source semantics, the activities that reflect the state of the source objects or involve communication with the database objects. Communication activities include modifying the database objects and relaying the user's requests to the application. They also include obtaining semantic information about the current state of the source objects (state includes both attribute values and object composition) and schema information such as object types. This information is used by display actions to reflect object semantics

Actions within interactor behavior use semantic information to place some constraints on what values can be entered for a particular attribute. For example, the constraints might be imposed by restricting the motion of a slider or disabling certain items in a menu. In some cases, semantic information would be obtained only at generation time and used to initialize the interactor. In other cases, it is obtained during display execution, as restrictions on allowable values or types are changed.

Semantic information also includes constraints for valid object connections, which may be enforced by a display when a user makes requests for manipulating object composition. The display can convey these constraints by affecting an interactor's characteristics, e.g. whether they are enabled to accept input or where they may be moved. Information about the state of object connections is also used to adjust the display format appropriately.

4.1.3 Running Example and Chapter Overview

The remainder of this chapter discusses the spec classes and the roles that each fills in relation to the construction model. Also described are the features that are definable in a spec, as opposed to those that are handled by ODDS-Runtime and thus are not definable by the designer. The definable features are specified through a spec's attributes and relationships, which are called the spec's *elements* (distinguishing them from the attributes and relationships of source objects). Names of spec elements are written in a teletype font, e.g., `elementA`.

In this and following chapters, Outline names are denoted by bold font; an Outline name is sometimes used as a synonym for the Outline itself. An Outline named **MealChoices** will be used as a running example for illustrating many of the spec classes. This Outline describes the display for assigning Meals to a Diet schedule, pictured in Figure 2.3 (Section 2.2). The two main functions performed through the meal-assignment display are:

- The user can create a connection from a DayPlan (within a Diet's schedule) to a Meal. The connection represents either the breakfast, lunch or dinner relationship of a DayPlan, depending on which relationship is currently displayed. To create a connection, the user clicks on a DayPlan display, then on a Meal display.

- The user can shift between viewing the breakfast, lunch, or dinner relationships. Each time the user clicks on the title bar, the display switches to the next relationship, using the ordering: breakfast, lunch, then dinner. When a transition is made, three parts of the display change. One is the relationship name presented in the title bar, and a second one is the set of Meal displays from which to choose. The list of Meals being displayed are those returned by one of the messages for obtaining meal choices: either "getBreakfastChoices", "getLunchChoices" or "getDinnerChoices" (defined in `Diet`). Thirdly, the lines between DayPlan and Meal displays change to reflect the connections for the currently displayed relationship.

In subsequent discussion, diagrams are used to illustrate the contents of Outlines and spec fragments. The diagrams provide a visual aid for understanding an Outline and how it parts are interconnected. In the diagrams, each kind of spec is represented by a certain graphic notation. A spec shown in a diagram is sometimes given a label, shown as underlined text. Labels simplify a diagram by denoting a spec that is drawn elsewhere, either in another part of the diagram or a separate diagram.

The spec classes are broken down into three general categories: those for building up Outlines, for describing display appearance, and for describing behavior. The following sections discuss each category.

4.2 Outline-Building Specs

An Outline spec is a display specification targeted for objects of a particular class. Several Outlines may be associated with a given class, allowing its instances to be displayed in different ways. An Outline has a `name` element that is used when the Outline is invoked by an application program or another Outline. An Outline's `layout` element is a Layout spec describing the display's initial image. For the most part, the Layout spec has a tree structure that parallels the nesting of display subcomponents. An Outline's `behavior` element is an array of Interaction specs. A display's behavior is viewed as a collection of separate behaviors attached to various parts of the display image, thus the Interaction specs are not typically in a single hierarchy that parallels the Layout structure.

An Outline may also have a `params` element, which holds a list of named Parameter specs (parameter names will be written in a slanted font). Parameters support *Outline generalization*, allowing certain details in the Outline to be left undefined until it is invoked; thus the display generated from an Outline can vary to suit different contexts. Generalization is useful in situations where several displays have a common framework, and differ only in certain details. For example, in the display of nutrition status (Figure 2.2), the bar displays for protein, fat and carbohydrates are basically the same, but differ in the nutrient that each presents. Therefore, the Outline describing the generalized bar display, called **NutrientData**, has a *nutrient* parameter that determines the desired path name, and is set when **NutrientData** is invoked. Within **NutrientData**, the spec that defines the bar display's source path has a parameter reference to *nutrient*.

Parameters are also used when defining a requirement that several format changes should occur simultaneously. These changes are coordinated by defining their triggering conditions to depend on a common parameter. An example of coordinated format changes is seen in the display for assigning the Meals to a Diet's schedule, which is discussed in detail at the end of this section.

A parameter is referenced from within the layout and behavior descriptions by using a

Symbol[1] that contains the parameter name. A parameter reference is used as a *trigger* or a *constant*, depending on where it is placed. The reference is a trigger if changes in the parameter value are used as a basis for choosing between alternative display formats. Otherwise, the reference is seen as a constant, meaning the parameter value is not tracked during display execution. The value is obtained only once, when supplied at an Outline invocation.

The specs discussed below do not describe a display activity, but are used throughout the layout and behavior descriptions to supply various kinds of information. Path specs serve in defining how a display uses data from its source object. Deferment and Iteration specs provide a way to specify *Outline composition*, i.e., where an Outline designates another Outline as the description for a display subcomponent.

The purpose of a Path spec is to refer to a specific part of a source object. As stated in Section 3.1, display specifications need this capability to identify where (in the source object) to obtain values that affect the display's creation and operation. Recall that a path is relative to a root object and is defined as a sequence of unary messages, e.g. #(currentNutrition fat) is a path relative to a Diet object. (The notation for a Path spec is the same as for a path.) The empty path, denoted #(), can be used to refer to the root object.

A Deferment spec describes the incorporation of one Outline into another. An Outline containing a Deferment is called a *deferring Outline*. One element of a Deferment is a `subOutline` that names the Outline being incorporated, called the *deferred Outline*. A Deferment also has a `subSource` element whose value is a Path indicating the source object for the deferred Outline. Since the `subSource` is specified as a Path, the subdisplay generated from a Deferment is associated with a position within a root object, not with a particular object. Thus, if the value at that position is updated during display execution, the new value in that position becomes the subdisplay's source object.

Since the `subOutline` element is an Outline name rather than a reference to an Outline spec, it does not completely determine what the deferred Outline is. Rather, the deferred Outline is identified at generation time, and the choice is based on the class of the subdisplay's source object, in addition to the `subOutline` element. Because this choice is delayed until generation of the deferring Outline, the generated display can be tailored to the state of each source object

[1] A Symbol is similar to a String, but represents a special token rather than simply being an array of characters. Thus, the characters in a Symbol cannot be changed. A Symbol is denoted by a pound sign followed by the Symbol's text, e.g., #aSymbol.

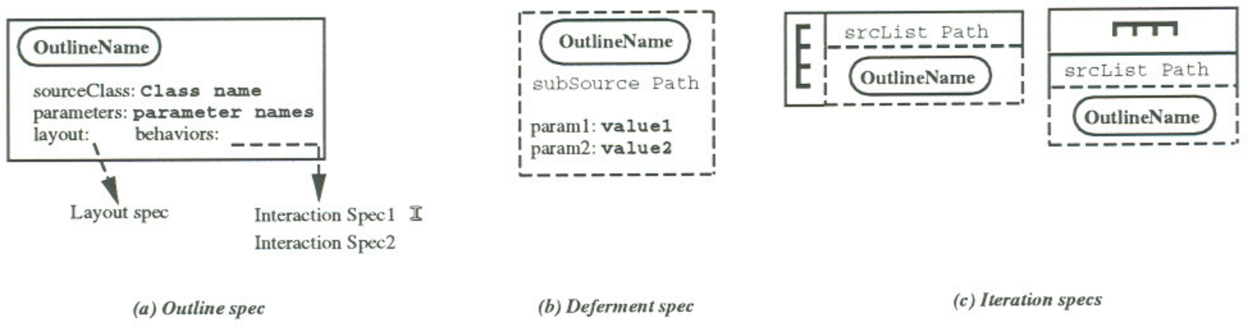


Figure 4.2: Outline, Deferment, and Iteration Notations

for which the Outline is invoked. For example, suppose there is an Outline for `Diet` named **DietSummary**, containing a Deferment spec with `subOutline` **AllNutrients** and `subSource` #(currentNutrition). Also suppose there are two Diet objects where the currentNutrition is a NutritionLog in one Diet, and is an ExtNutritionLog in the other Diet. If DietSummary is invoked for both Diets, the choice for the deferred Outline will differ when generating the two displays, provided that an Outline named AllNutrients exists for both `NutritionLog` and `ExtNutritionLog`.

The dynamic binding of deferred Outlines also increases the opportunity for reusing Outlines after schema changes. Referring to the example above, when subclasses are defined for `NutritionLog`, the DietSummary Outline does not need to be modified to display Diets referring to instances of the new subclass. All that is needed is to define an Outline named AllNutrients that is specialized for the new subclass.

An Iteration spec describes a display for a list of objects, where each member in the list is displayed using the Outline named in the Iteration's `membOutline` element. The member displays may be placed in a row or column, as specified by a `configuration` element. The `sourceList` element holds a Path spec that leads to the collection of objects to be displayed. Finally, the `isDynamic` element indicates whether the display generated from the Iteration should change its format when members are added to or deleted from the display's source object.

Diagram Notations

The diagram notation for an Outline (Figure 4.2a) presents its source class, parameters, and diagrams of its Layout and Interaction specs. The **I** symbol marks *initialization behaviors* that must be executed when a display is first generated. A Deferment's notation (Figure 4.2b) shows

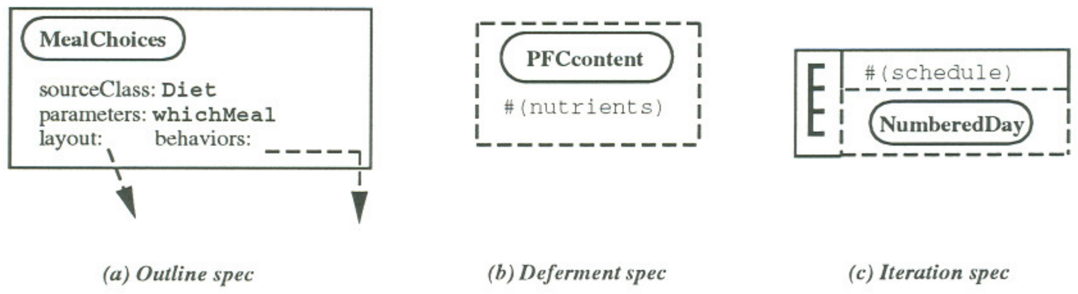


Figure 4.3: Diagram Examples

its `subOutline`, `subSource`, and the parameter values submitted to the deferred Outline, if any. An Iteration's notation presents values for the `sourceList` and `membOutline` elements as shown in Figure 4.2c. The notation on the left represents an Iteration with a column `configuration` for member displays, while the one on the right indicates a row `configuration`. A parameter reference (not shown) is denoted by a Symbol consisting of the parameter name prefixed by the letters PAR.

Diagrams for some spec objects are shown in Figure 4.3. Figure 4.3a is the diagram notation for **MealChoices**, which has `Diet` as it source class and has a parameter *whichMeal*. This parameter determines whether the breakfast, lunch or dinner relationship is being displayed, and it coordinates the three format changes that occur when switching between relationships. (The specs that make up the `layout` and `behavior` elements are described in later sections.) The Deferment in Figure 4.3b is one that might be used to represent a subdisplay for the nutrients in a Meal. The subdisplay is described by an Outline named **PFCcontent** (Protein, Fat, and Carbohydrate content). The Iteration in Figure 4.3c describes the column of DayPlan displays in the meal-assignment display; each DayPlan display is generated from an Outline named **NumberedDay**.

4.3 Specs for Display Appearance

Layout specs define the image presentation portion of a display description. The `features` element within a Layout spec is a VisFeature spec defining the graphical details (such as colors or fonts) that apply to the Layout spec. The graphical features are defined as a distinct object from a Layout spec so that they may be shared among several Layout specs. Section 4.3.1 discusses Layout specs and how they are composed. Section 4.3.2 discusses the graphical features definable in ODDS and how these definitions can be shared.

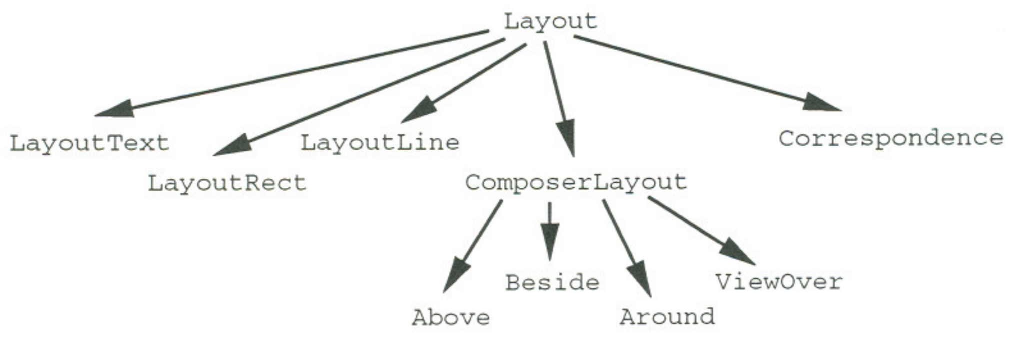


Figure 4.4: Types of Layout Specs

4.3.1 Layout Specs

Figure 4.4 shows the different kinds of Layout specs and the subclass relationships between the Layout classes. In the ODDS prototype, the primitive types in images are text strings, lines, and rectangles, where each type is defined with a different kind of spec. LayoutLine specs are intended for describing connections between images rather than being primitives for creating polygons.

The elements in the primitive Layout specs are as follows:

- A LayoutText has a `string` element that holds either a character string to be displayed or a Path spec defining where to obtain a character string. A LayoutText also has `width` and `height` elements defining the space allocated for displaying text. These elements allow the designer to leave room in the display if the displayed string is expected to change during execution.

  A word that extends beyond the specified `width` is placed on the following line. If `width` is left unspecified, it is assumed to be the width of the displayed string. If the `string` value is a Path spec, the displayed string is the path's value at the time a display is generated. The LayoutText's `height` is the maximum number of lines of text that can be displayed. If `height` is not specified, space is allocated for the number of lines needed to draw the displayed string within the specified width. Therefore, if only the `string` element is defined in a LayoutText, the default amount of space allocated is the amount needed to draw the displayed string as a single line of text.

- A LayoutRect spec has `height` and `width` elements. The rectangle image is further defined by the color and border features in the LayoutRect.

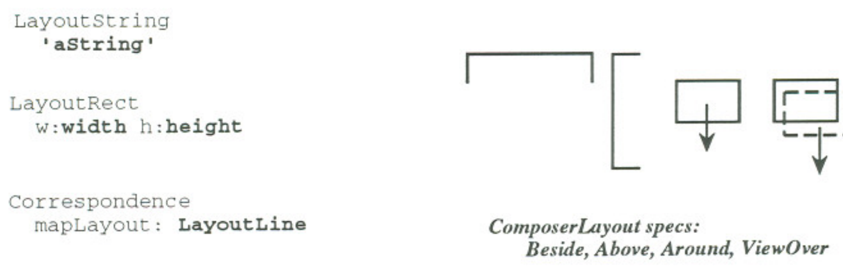


Figure 4.5: Layout Spec Notations

- A LayoutLine has `start` and `end` elements that define the two Layout specs whose images are connected by a line. LayoutLine specs have a default positioning policy for placing the endpoints on connected images. As long as the images of the endpoint Layouts are beside each other (i.e., their extents in the x-direction do not overlap), the line connects the two closest sides at their centers. If the images are not beside each other, the line is drawn from the top center of the lower image to the bottom center of the other image. In the current prototype, a LayoutLine is used only in the context of a Correspondence spec.

Various kinds of ComposerLayout specs describe how the primitives are combined to build up a complex image. All kinds of ComposerLayouts have a `subparts` element that is a collection of Layout specs; thus Layout specs have a hierarchical structure and can be nested to any depth. A ComposerLayout spec is said to be the *parent* of the Layout specs in `subparts`. The notations for the primitive Layout and ComposerLayout specs are shown in Figure 4.5.

Specific kinds of ComposerLayouts include Above and Beside specs that define how the display image is spatially composed from the `subparts` specs. The ordering of the specs in `subparts` defines the visual ordering in the generated image.

An Around spec represents a border surrounding the image described by the spec's `subparts`. The `subparts` element of an Around spec contains at most one item. A border's size in the generated display depends on the size of the contained image and the spacing defined for the Around spec. (The following section discusses how spacing is defined.) However, a minimum size can be defined in the Around spec's `minBounds` element. An Around spec also has a `shape` element, chosen from a designated set of keywords representing the types of borders that may be drawn. The choices available depends on the capabilities built into the runtime system. The ODDS prototype supports rectangular borders only.

A ViewOver spec defines a *viewport*, a rectangular area of fixed size that will hold an image defined by another Layout spec. A ViewOver models constraints on the screen space allocated for the enclosed image; thus the image can change size without affecting other parts of the display. A ViewOver also defines the image's position relative to the viewport, and thus plays a part in describing scrolling behavior. (See MotionOp in Section 4.4.1.)

A Correspondence spec describes the visual representation for connections between source objects. Correspondence specs provide an alternative to the customary way of expressing connections, where the display of the referring object contains the display of the object being referenced.[2] A Correspondence defines how to display a certain relationship for each source object in a collection; the objects in this collection are called the *domain* source objects. For a particular relationship `R`, the relationship value of a domain object is called its *range object with respect to* `R`. For example, the meal-assignment display in Figure 2.3 presents a collection of DayPlan objects, and presents each DayPlan's connection for the 'breakfast' relationship. The range objects are the Meals to which the DayPlans are connected. Essentially, the display features described by a Correspondence represent a set of connections that could be viewed as a mapping from a domain collection to a range collection.

The visual representation for a connection is specified in the `mapLayout` element of a Correspondence spec. For a line representaton, `mapLayout` is set to a LayoutLine spec. Two special Layout specs, called `domainPlace` and `rangePlace`, are used within the `mapLayout` to act as place-holders for the actual Layout objects that represent the related display images.

Another possibility for visually representing connections is to juxtapose the domain and range displays in some way, however this expressiveness is not available in the ODDS prototype. If a Beside or Above spec were usedas the `mapLayout`, the displays of the related objects would be positioned accordingly.

The display images involved in a Correspondence are defined by a a list of Layout specs representing the domain objects and the database relationship being represented by the Correspondence. Alternatively, an Iteration spec could be used instead of a list of Layout specs. The relationship name is defined in an Interaction spec that is associated with the Correspondence (see DBRelConnect in Section 4.4.1). It is up to the display system to determine which display images represent the range values for the message.

[2]Although a Correspondence spec refers to other Layout specs, it is not a kind of ComposerLayout since it does not define positioning.

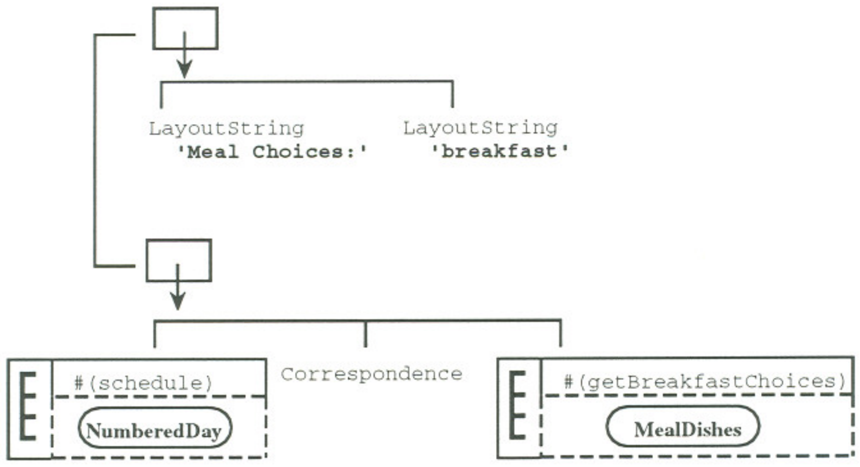


Figure 4.6: Example of Layout Specs

Figure 4.6 shows a diagram for one possible display image of the meal-assignment display. The Layout spec contains two Iterations. One describes the column of DayPlan subdisplays that represents the Diet's schedule and uses **NumberedDay** as its `subOutline`. The other Iteration describes the column of Meal subdisplays representing a list of Meal selections using **MealDishes**. The displayed list is the value for the path #(getBreakfastChoices) within the Diet object. A Correspondence describes the set of lines connecting subdisplays in the two columns, representing the connections between DayPlan and Meal objects.

4.3.2 Specs for Visual Features

When executing drawing operations, some graphics systems maintain a *graphics context*, a data structure whose contents determines what graphical attributes apply to the current drawing function. Systems that use graphics contexts (or a similar concept) include the X11 Window System, MS Windows, and Macintosh's QuickDraw. For a programmer using the graphics system, a graphics context provides the advantage that the attribute information need not be supplied with each call to a drawing function. The programmer updates only those graphics-context values that vary from one function to the next.

VisFeatureSet specs represent the graphics context state for rendering a particular image. A VisFeatureSet contains five categories of features: spacing, color, fill pattern, border, and text-related features. Each category is modeled as a sub-object within a VisFeatureSet.

A VisFeatureSet spec does not necessarily hold values for all definable features. The display designer need only define the features that differ from those defined in a Layout spec's parent.

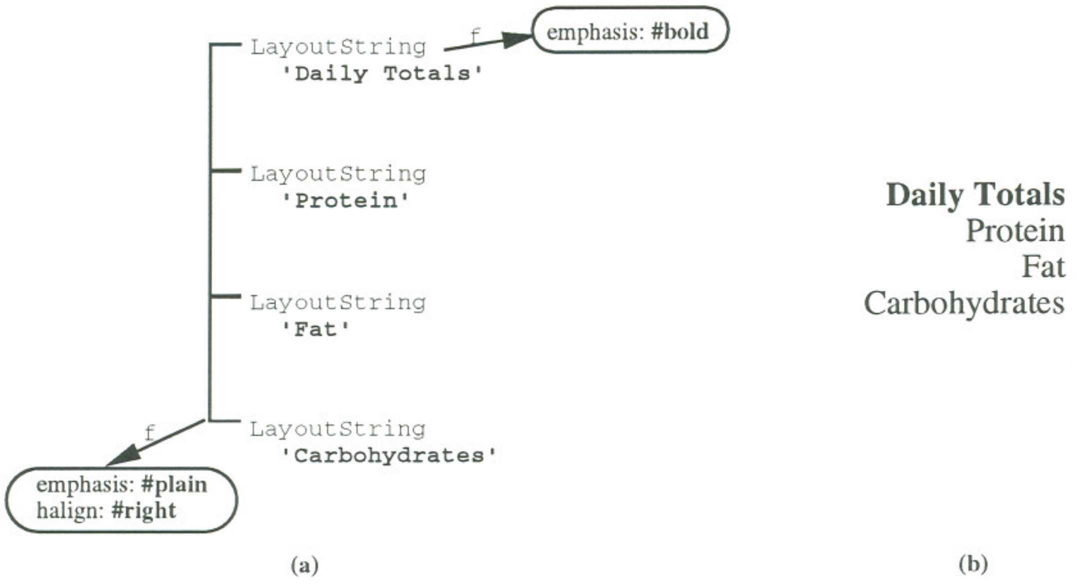


Figure 4.7: Example of Shared Features

In other words, a feature is shared between a ComposerLayout spec and its subpart if left undefined for the subpart. In the case of a root Layout spec (the one referenced by an Outline), features that are left undefined are assigned some default values. As an example of shared features, consider the Above spec in Figure 4.7a, which describes the lines of text in Figure 4.7b. Since all but the top line have a plain text style, the text emphasis can be defined in the Above spec's `features` rather than in each LayoutString. The emphasis is defined in the LayoutString for the top line, making the description more concise.

The feature sharing described above applies to the initial rendering of the display image. However, the shared features are not constrained to remain consistent with each other while the display is running. To specify that features should be shared at runtime, the elements of the VisFeatureSets must refer to the same spec, as shown in Figure 4.8. In this example, the Layout spec describes the image for a screen button that is sometimes highlighted by reversing its foreground and background colors. In either state, the text background and the background inside the border should have the same color.

The elements in each of the five feature categories are described below:

- **Spacing Features.** Horizontal and vertical spacing are defined through the elements `hspace` and `vspace`. Currently, spacing is defined in terms of inches. Ideally, one would be able to set these elements in terms of various units, such as inches, centimeters, or

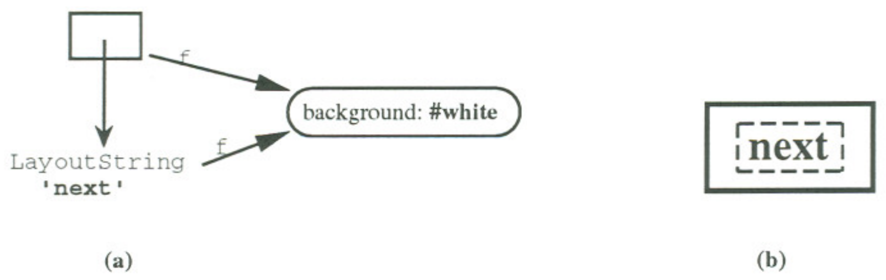


Figure 4.8: Example of Constrained Features

pixels, depending on which is most suitable. The values for `hspace` and `vspace` have different effects depending on the Layout spec being addressed. In a Beside spec, `hspace` specifies the distance between the subparts and `vspace` does not have any significance. Likewise, in an Above spec, only the `vspace` feature is meaningful. In an Around spec, both features are used to specify the spaces between the subpart and the surrounding border.

- **Color Features.** The `foreground` color of an image is the color in which the text or graphics are drawn. The `background` color is the color used for the rest of the image, or the "empty space".

- **Fill Features.** The fill features apply to the background of an image, which may be a solid color or a tiled pattern. This choice is specified in the `fillStyle` element. The `pattern` element holds the bitmap that will be used if the background is tiled.

- **Border Features.** A border refers to the bounding shape specified in an Around spec, or to line boundaries between the subparts of a Beside or Above spec. The `borderwidth` holds the point size for borders, and its default value is 0 (no border). Like an image background, a border is either solid or tiled. The elements `bordStyle` and `bordPattern` specify these border characteristics.

- **Text Features.** Features for text include a `fontname` and an `emphasis`, which may be plain, bold, or italic. In the current implementation of ODDS, font size is defined as part of the `fontname`, e.g., "Serif12".

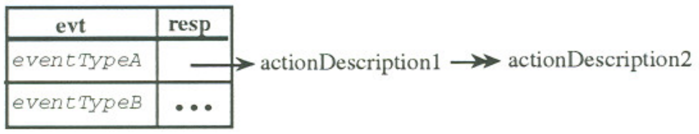


Figure 4.9: Notation for Event Mappings

4.4 Specs for Display Behavior

The display activities for action sequencing and semantic support make up the behavior for a display. The Interaction specs of an Outline cover the specification of these activities.

In general, action sequencing is described as an *event mapping* from *event types* to *response descriptions.* An event type may represent the arrival of user input or a change in a source object. It may also represent an *internal event*, which is a significant condition that arises during event processing and needs to be communicated from one behavior to another. For example, the selection of a DayPlan subdisplay in the meal-assignment display is a significant condition because the selected DayPlan must be recorded for use at a later time. An EventType spec describes and event type; it consists of a type name and a list of names for data values that will accompany events of that type.

A response description defines the action or sequence of actions that will be invoked when certain conditions occur. Often the occurrence of a certain type of event is the only condition needed before invoking an action, but at times further conditions are required, as discussed in Section 4.4.1, under *Flags and Data Variables.* The event processing underlying an event mapping is as follows. All th actions defined in the responsed description are executed before processing any other events originating from the user or the database. However, if the event response includes raising an internal event, that event its processed immediately.

Figure 4.9 shows the diagram notation for event mappings. Each row in the table holds an event type's name and its associated response description. Double-headed arrows indicate the sequence of action descriptions within a response description. Ellipses are used to represent a response description that is not elaborated in a diagram.

The description of a display's action sequencing is distributed among the Interaction specs of an Outline, where each spec holds an event mapping. The types of Interaction specs are shown in Figure 4.10. Table 4.1 shows the types of specs associated with image presentation and different parts of action sequencing. ImageOp, MotionOp, and Router specs describe

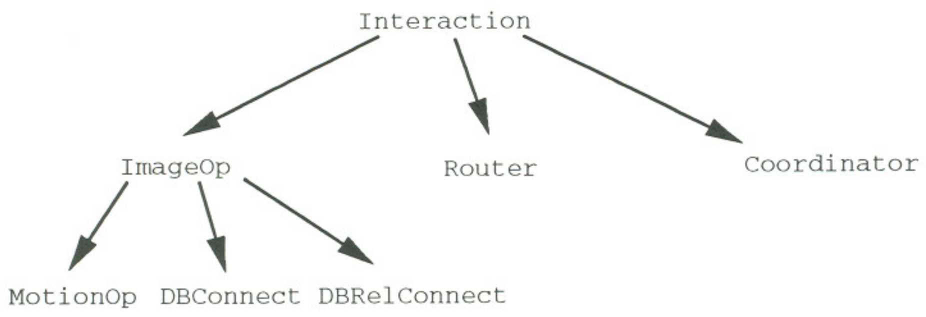


Figure 4.10: Classes of Interaction Specs

Activity	*Specs used*
Image Presentation	Layout, LayoutText, etc. VisFeatureSet
Action Sequencing	
Interactor Behavior	
Process user input	ImageOp, MotionOp, Router, DBConnect, or DBRelConnect
Process database events	DBConnect or DBRelConnect
Interactor Management	Coordinator

Table 4.1: Display Activities and Associated Specs

interactor behavior, Coordinator specs describe interactor management, and DBConnect and DBRelConnect specs describe semantic feedback and communication with the database. Since a DBConnect or DBRelConnect spec inherits elements of ImageOp specs, it can combine the description of input processing and the behavior necessary to represent the state of a source value (i.e., obtaining the current value of a source object and sending messages to change the value according to user input).

The relationships between parts of the construction model are reflected in the composition of Interaction specs:

- Non-semantic and semantic actions both make use of image presentation, as indicated by the dashed arrows in Figure 4.1. An Interaction spec that describes display-changing behavior refers to a Layout spec, which is the Interaction spec's `subject` element. Response descriptions within the Interaction spec's event mapping describe updates to the subject, thus representing the changes that will take place in the display image at runtime.

- The arrow from interactor management to interactor behavior is reflected in a Coordinator, which refers to a group of Interaction specs making up its `subInteraction` element, among which communication is described.

- Arrows from interactor behavior to semantic support are reflected in a DBConnect or DBRelConnect spec that can contain descriptions of semantic actions in their event mappings. In general, non-semantic and semantic actions both are described using a MatchMaker spec, as will be discussed in Section 4.4.2. However, descriptions of semantic actions are found only in an event mapping of a DBConnect or DBRelConnect spec because only these specs can define access to semantic information.

The following subsections provide more detail on how behavior is described in Interaction specs. Recall that in the construction model, display behavior is broken down into action sequencing, non-semantic actions, and semantic actions. Section 4.4.1 discusses the description of action sequencing, and Section 4.4.2 discusses specs for describing actions. Section 4.4.3 discusses the specs for describing format changes, which differ from other action descriptions because they involve specs outside of an event mapping.

4.4.1 Describing Action Sequencing

Several UIMSs [Green85b, Jacob86, Hill86, Sibert86, Bass90, Hudson88] adopt the view that a user interface behaves as a collection of event handlers, and thus they describe action sequencing based on events and responses to events. An event-based description that is large can be difficult to understand, since the control flow it describes becomes unclear. To avoid this problem, the description of action sequencing in an Outline is partitioned into several event mappings, where each is associated with a logical unit of behavior, i.e., an Interaction spec. Because the Interaction subclasses model different aspects of action sequencing, event mappings are different in each kind of Interaction spec. We first describe things common to all Interaction specs, then describe the differences in their event mappings. Further details on the elements of Interaction specs are then provided.

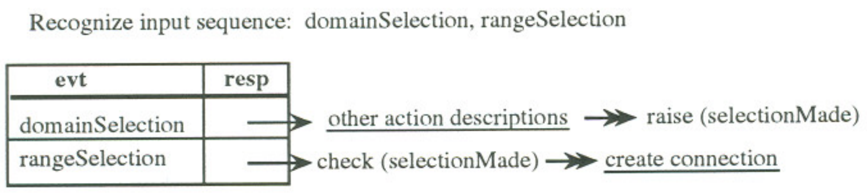


Figure 4.11: Event Mapping Using Flags

Flags and Data Variables

An event mapping must be able to express that action invocation requires the occurrence of several events rather than just a single event. For example, in the meal-assignment display, a connection from a DayPlan object to a Meal is created only after the user has performed two steps: the user must first select a DayPlan display by clicking on it, then select a Meal display in a similar fashion. To describe the recognition of event sequences, a set of *flags* may be defined for an Interaction spec. A flag is a boolean variable; the collective state of the flags during display execution represents the control state for an Interaction executor. The flag concept was introduced in the Sassafras UIMS [Hill86], which has a specification language developed specifically for describing multiple concurrent dialogues.

A flag can be used to record when a certain point has been reached within a desired input sequence. Raising a flag or examining a flag's state are actions that can be defined as part of a response description. When a flag is examined, the subsequent actions defined in the response description will be executed only if the flag is raised. To describe recognition of an event sequence $e_1,...,e_n$, a flag is associated with each event type e_i in the sequence, and a raise action is made part of e_i's response description. In addition, the response description defines a check on the flag for e_{i-1} before raising e_i's flag. As a simple example, the event mapping in Figure 4.11 describes recognition of the input sequence for creating a DayPlan-to-Meal connection. The flag *selectionMade* is associated with the first event type, and the response description for the second event type defines a check on *selectionMade*.

In some cases, the ordering of required events is not relevant when triggering an action; the events merely have to occur at some time before the action can be invoked. To express this synchronization, multiple flags are used as the condition for triggering the action. For these cases, event mappings provide a more concise expression than state-transition diagrams. All the possible orderings of valid event sequences would be modeled explicitly in such a diagram, resulting in exponential growth in the number of states with respect to the number of expected events. The ability to express synchronization is especially useful in Coordinators, for defining interactor communication independently of the behaviors within interactors.

The flags defined for an Interaction spec are not accessible to any other Interaction spec; thus the behavior defined in one Interaction spec cannot be dependent on the state of some flag defined in another Interaction spec. Rather, dependence among behaviors occurs only through internal events, and the description of any dependencies is captured in one or more Coordinator specs. Keeping the flags' scope local to a single Interaction spec helps to maintain clarity of their significance in the display behavior; i.e., what conditions result in raising a flag and what is affected by changes in the flag's state. Without such localization, the meaning of a behavioral description is easily obscured.

In addition to describing control state through flags, an Interaction spec can describe some data state associated with the described behavior. Data state related to the execution of a generated display does not belong in the database since such state concerns a source object's presentation, not its behavior. Furthermore, this state is typically not of interest to the application. A set of `variables` can be defined in an Interaction spec. Each variable defined represents a named location that holds data values during display execution. The variable names are used within the response descriptions for the Interaction spec. Like flags, data variables are accessible only within a single Interaction spec.

The meal-assignment display provides an example of behavior that uses stored data. Recall that clicking on the title bar modifies the relationship being displayed. Since the display cycles through the relationships in a particular order (breakfast, lunch, dinner), determining what relationship to display next depends on which one is currently being displayed. Thus, the name of the current relationship is kept in a variable and updated whenever there is a switch.

The diagram notations for Interaction specs are illustrated in Figure 4.12. A spec is represented by its class name, followed by its variable names (if any), listed in parentheses. As shown in Figure 4.12a, the Interaction spec's `subject` is identified by two joined lines. The names of event types, flags, and variables are written in an italic font. Any additional elements present in a particular kind of Interaction spec are shown under the spec's class name. The spec's event mapping may or may not be shown in a diagram. If shown, it appears above the class-specific elements. The notation for Router and Coordinator specs indicates their `subInteraction` elements, as shown in Figure 4.12b.

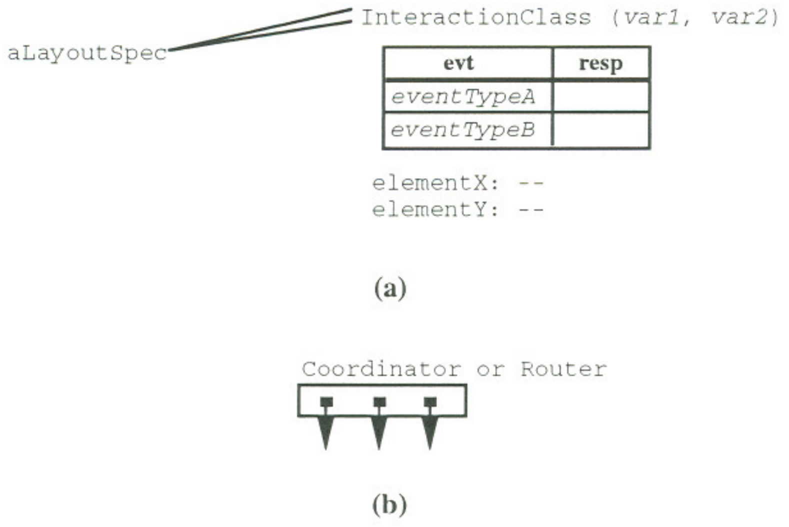


Figure 4.12: Notations for Interaction Specs

Differences in Event Mappings

The event mappings differ in each of the Interaction subclasses since they model different aspects of action sequencing. Each kind of Interaction spec is discussed below.

ImageOp. An ImageOp spec describes the behavior of an interactor such as a menu button or gauge. The types of actions defined in an ImageOp spec include updating a Layout executor to alter the display image, updating flags or variables, or generating internal events that will be forwarded to another Interaction executor. These actions can be defined in the other kinds of Interaction specs as well. Predefined event types that are associated with physical device input include those for button presses and clicks, and for detecting when the cursor has moved into or out of an image.

MotionOp. A MotionOp spec describes interactor behavior for moving an image on the screen. For example, a slider, a scrollbar and a scrolling view all have behavior that involves image movement. The MotionOp class abstracts the definition of such behavior, so the designer needs to specify only certain key characteristics rather than all the actions involved in moving an image. Among the characteristics to be defined are the means for obtaining the destination point and a reference point on the image being moved; both pieces of information are needed to define where the image is placed. The destination point may be the cursor location or a specific coordinate stored in an Interaction variable. If the destination is defined as a coordinate, a rectangular screen that acts as a frame of reference is also defined. Behavior abstractions such as MotionOp permit a variety of behaviors to be described without requiring the definition of events or actions that are common to all. Similar abstractions have been developed in the Garnet development environment [Myers90], and could be modeled in the specification framework as subclasses of ImageOp.

Router. Router specs define which areas of the display image react to user input. A Router's `subject` Layout defines its area of interest on the screen. A Router has one or more `subInteraction` elements that are ImageOp specs (or specs from any subclass of ImageOp). These specs represent the behaviors that can be activated by user inputs directed at that area.

DBConnect and DBRelConnect. The Interaction specs discussed so far define activities for input processing only. The sequencing defined in DBConnect and DBRelConnect specs can involve semantic actions. DBConnect specs model behavior that either reflects the state of source objects to the user or initiates operations on source objects. A number of predefined event types represent the occurrence of source object changes in the database objects and define what information is supplied from the database when changes occur. To reflect source changes, the response descriptions in DBConnect specs define how the information in those events is used to update the display image. A response description can also define an action of sending a message to a source object, either to query or update the object.

A DBRelConnect spec is the behavioral counterpart for a Correspondence spec. As stated earlier, a DBRelConnect defines what relationship is visually represented by the Correspondence. In addition, a DBRelConnect's event mapping can define any display changes or other activities that occur in response to a change in the specified relationship.

Coordinator. A Coordinator spec defines communication and data transfer that occurs among interactors during display execution. It can be used to describe behavior of a composite interactor, e.g., a panel of radio buttons. A response description for an internal event type typically indicates where to forward events of that type. However, the response description can include the kinds of action descriptions allowed in other kinds of Interaction specs.

A `subInteraction` spec in a Coordinator may be a kind of ImageOp, an Iteration, or a Deferment spec. When an Iteration or Deferment spec is a `subInteraction`, it represents the `behavior` element of the deferred Outline (or multiple instances of the Outline, in the case of an Iteration spec). The following example illustrates a Coordinator that has Iterations as `subInteraction` elements, and also illustrates use of a DBRelConnect.

The behavior being defined in Figure 4.13 is to process user input for assigning a Meal

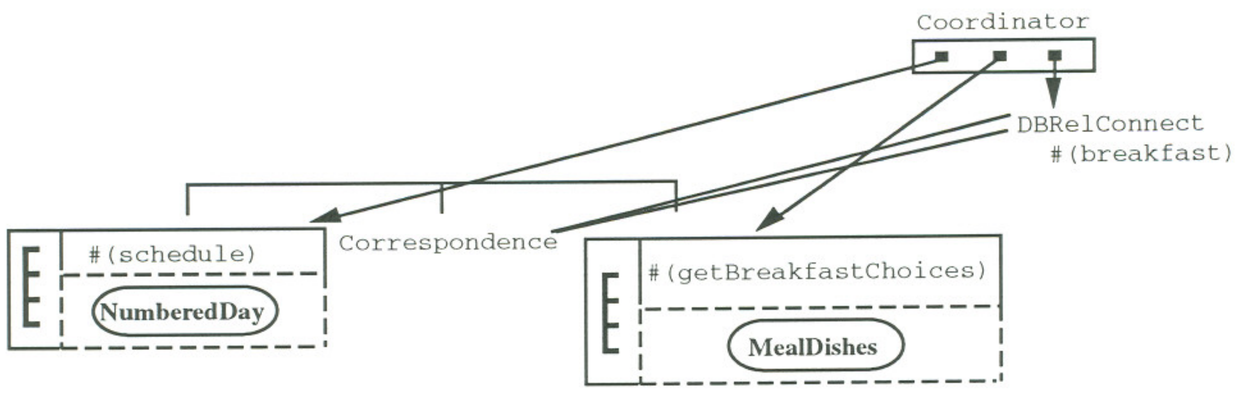


Figure 4.13: Example of Coordinator and DBRelConnect Specs

to a DayPlan. Recall that the user must click on a DayPlan display and then on a Meal display to create a connection between their source objects. The Coordinator in the figure defines what happens to internal events produced from any of the DayPlan or Meal subdisplays. Both **NumberedDay** and **MealDishes** Outlines define behavior to create an InternalEvent whenever a mouse button is clicked within the display image. In **NumberedDay**, the created event has the type *domainSelection*, while in **MealDishes**, it has type *rangeSelection*.

The Coordinator's event mapping was discussed in an earlier section, and is partially depicted in Figure 4.11. The action descriptions denoted by 'store selection' and 'create connection' are references to the DBRelConnect spec. Thus, all internal events are forwarded to the corresponding DBRelConnect executor, which takes care of storing the selection and creating the source object connection. These activities are defined in the event mapping of the DBRelConnect spec.

Elements of Interaction Specs

To recap, all Interaction specs include the following elements:

- `subject`, a Layout spec,
- `eventMap`, a mapping from event types to response descriptions,
- `flags`, the names of flags used in `eventMap`, and
- `variables`, a list of variables used in `eventMap`.

Aside from ImageOp, all Interaction classes define additional elements for specs, and some classes have a special significance for elements named above.

In a MotionOp spec, the `subject` represents the image being moved. The MotionOp's `eventMap` may contain the symbol #motion as an action description, signifying that the `subject` Layout's image should be moved. A MotionOp includes these additional elements:

- `boundLayout` may be either a LayoutRect or ViewOver spec. It provides a frame of reference for defining where the `subject` should be placed.

- `follow`, which indicates where to obtain the *guide coordinate* that determines placement of the image. The guide coordinate comes either from the mouse cursor, or from the `layoutPosition` element in the MotionOp spec.

- `layoutPosition`, whose value must be a proportional coordinate relative to the size of the `boundLayout` image. For example, the coordinate (0,0) represents the top left corner of the `boundLayout`, (1,1) is the bottom right corner, and (0.5,0.5) is the center. When `layoutPosition` is defined a spec, it indicates the initial position of the movable image. Typically, action descriptions in the MotionOp's event mapping define calculation of the guide coordinate, which is then set as the value of `layoutPosition`.

- `refPt` defines the point on the movable image that will coincide with the guide coordinate. The allowed values are Symbols that indicate either an edge reference (e.g., #top, #bottom) or a corner reference (e.g., #topleft). Specifying an edge reference means that the image is constrained to move only in one dimension, to keep that edge in line with the guide coordinate. For example, a scrollbar or slider exhibits one-dimensional movement. If a corner reference is used, the image can be moved vertically or horizontally. For example, an image within a viewport can be scrolled in both directions using scrollbars, or might be positioned according to the location of mouse clicks.

A Router spec holds `subInteraction` specs, as stated earlier. The `subject` Layout of each `subInteraction` must be either the same as or a subpart of the Router's `subject`. The event mapping of a Router is derived from the event mappings in its `subInteraction` specs. The set of event types in a Router's mapping is the collection of event types from the `subInteraction` specs. The response description for each event type is the `subInteraction` spec whose mapping contains that event type. In the ODDS prototype, an event type can appear in the mapping of only one `subInteraction` spec, so that at runtime only one behavior will be activated per event.

In a DBConnect spec, the `sourceProxy` element defines a path in the source object from which values are taken and are used by the display during execution.

In a DBRelConnect spec, the `sourceLink` element defines a relationship being represented in the display. The Symbol #relUpdate is used as an action description that represents an update to the visual representation of the displayed relationship. Typically, this action is specified in response to a database event signifying that a domain object has been updated with a new range value. The display change that reflects the updated connection is determined by the `mapLayout` element of the Correspondence, and is handled automatically by the runtime system.

In a Coordinator spec, the `subInteraction` specs represent executors that communicate during display execution. An action to forward an internal event is described by a reference to the `subInteraction` spec that represents the event's destination (as seen in the Coordinator example presented in the subsection *Differences in Event Mappings.*).

4.4.2 Describing Actions

The actions defined in response descriptions include semantic and non-semantic actions. In addition, response descriptions define some activities for managing sequencing, such as modifying flags or sending internal events. In Table 4.2, the left column lists the various functions performed in response to events, as identified in ODDS' construction model. The right column states what kinds of specs are used to describe each function. Performing the activities listed in the table basically requires updating Layout executors or variables in Interaction executors, or sending messages to objects in the database.

The basic mechanism for describing update operations and message sending is the MatchMaker spec. The concept of a MatchMaker originated from research to model database operations (commands and queries) in a non-behavioral database model for complex objects [Zhu89]. In a MatchMaker spec, describing an action is broken down into two main parts: identifying the objects participating in the operation and describing the changes that result from the operation. Thus an action is modeled as a pattern-matching step followed by a data-manipulation step, hence the name MatchMaker.

At runtime, the objects participating in an action may be values from an Interaction or Layout executor, or they may be data from the event that triggers the action. In a behavioral

Activity	*Specs used*
Non-Semantic Actions	
Reflect user input	MatchMaker
Generate internal events	MatchMaker and InternalEventType
Format changes based on user events	ChoiceMap and (ParamDesc or FormatDesc)
Syntactic feedback	MatchMaker
Create related displays	MatchMaker and SpawnDesc
Semantic Actions	
Reflect changes in object composition	ChoiceMap and (ParamDesc or FormatDesc) Correspondence/DBRelConnect
Reflect changes to attribute values	MatchMaker
Validate user-input values and user requests for composition changes	MatchMaker and MessageDesc
Modify source objects	MatchMaker and MessageDesc

Table 4.2: Specs for Describing Actions

description, the participating values are identified using object templates that represent runtime objects. A template designated as the *matcher* is annotated with *object tags* that mark places of interest for the action. An object tag is considered to be bound to the runtime object represented by the place being marked. The participating objects for an action may also include a value resulting from some computation. Defining computations within a MatchMaker is discussed below.

The description of changes resulting from an action is accomplished through a template called the *maker*, which also is marked with object tags. The maker represents the runtime object(s) modified by the action. Object tags on the maker are a subset of those in the matcher. In the maker, the placement of an object tag represents the assignment of its bound value to place being marked.

As an example, the MatchMaker in Figure 4.14 defines an operation that reverses the foreground and background colors of a Layout executor. An underlined class name denotes an object template rather than an instance of the named class.

In the make template, the object tags **VF**, **FC**, and **BC** mark the paths #(features), #(features foreColor), and #(features backColor), respectively. The maker template defines the VisFeatureSet executor bound to **VF** as the object being updated. Specifically, the `foreColor`

```
Matcher:
Layout (features -> VisFeatureSet VF: ( foreColor -> FC: Object
                                        backColor -> BC: Object ) )

Maker:
VisFeatureSet VF: ( foreColor -> BC: Object
                    backColor -> FC: Object )
```

Figure 4.14: MatchMaker Example

and `backColor` fields of that VisFeatureSet executor will be updated with the objects bound to tags **BC** and **FC**, thus reversing the background and foreground colors of the Layout executor.

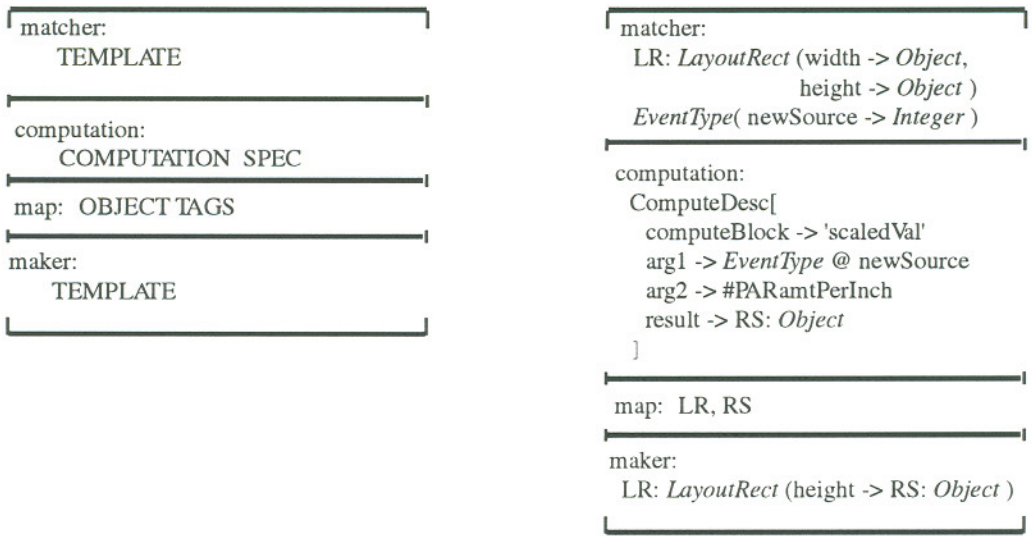
MatchMakers optionally contain a description of computations that use the participating objects as arguments. A ComputeDesc spec is placed in a MatchMaker to express update operations that involve arithmetic computation and conditionals, thus supplementing the matchmaking semantics expressed via the templates and object tags. The computation represented by a ComputeDesc cannot update any runtime objects; it merely returns a value. Supporting computation within MatchMakers allows procedural specification to be incorporated within the context of declarative specification. The specs define the point at which the computation is executed, and the computation itself is defined outside of the specification.

Elements in MatchMakers

This section describes the elements of MatchMakers and introduces other specs used with a MatchMaker when defining specific actions.

In a MatchMaker spec, the `matcher` element consists of one to three templates, depending on what pieces of data are needed for the operation being defined. The `matcher` includes an Interaction template if Interaction elements or variables are involved in the operation. If the operation updates the display image, a Layout template is included. If the operation uses data from the event that triggers the operation, the `matcher` includes an EventType template.

A MatchMaker also has a `map` element that defines the set of template associations represented by object tags. As explained earlier, an object tag marks certain sub-templates within the `matcher` and `maker` templates. The `map` has an association for each object tag, recording the two sub-templates that are marked by that object tag. The `maker` element consists of one or more templates, depending on what objects are to be updated by the described operation.



```
matcher:
    TEMPLATE
computation:
    COMPUTATION SPEC
map:  OBJECT TAGS
maker:
    TEMPLATE
```

```
matcher:
  LR: LayoutRect (width -> Object,
                  height -> Object )
  EventType( newSource -> Integer )
computation:
  ComputeDesc[
    computeBlock -> 'scaledVal'
    arg1 -> EventType @ newSource
    arg2 -> #PARamtPerInch
    result -> RS: Object
  ]
map:  LR, RS
maker:
  LR: LayoutRect (height -> RS: Object )
```

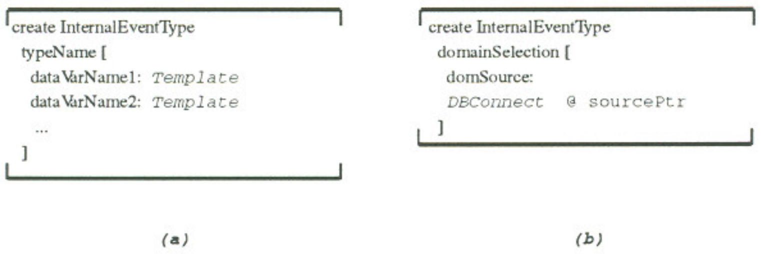
Figure 4.15: Diagram Notation and Example for MatchMakers

Any template sub-template in the `maker` that represents an updated object must have an object tag. The diagram notation for MatchMakers and an example diagram are shown in Figure 4.15. The MatchMaker shown describes an operation that adjusts the height of a rectangle image according to a value that accompanies a database event.[3]

A MatchMaker spec can have a `computation` element that holds different kinds of specs depending on the action being described. Referring back to Table 4.1, some actions that require specific specs are generating internal events, sending database messages, and creating related displays. The `computation` is a ComputeDesc spec when the described action is arithmetic or evaluation of some procedural code. An InternalEventType spec is used to describe creation of an internal event, a MessageDesc spec describes sending a message to a source object, and a SpawnDesc spec describes creation of another display.

A ComputeDesc spec represents a piece of procedural code. It has a `computeName` that defines a name (a string) associated with the block of code to be executed. The name and the code must be registered with the runtime system, where it is kept in a table called the Computation Library. The registered code must be written in Smalltalk; however this requirement does not restrict the choice of the application's implementation language because the runtime system is responsible for executing the registed code.

[3]The Outline containing this MatchMaker is discussed in Section 8.1.2.



```
create InternalEventType
  typeName [
    dataVarName1: Template
    dataVarName2: Template
    ...
  ]
```

(a)

```
create InternalEventType
  domainSelection [
    domSource:
    DBConnect  @ sourcePtr
  ]
```

(b)

Figure 4.16: MatchMakers for Creating Internal Events

The ComputeDesc also defines the values or the locations of the arguments to the computation block. An argument location is specified by a reference to a template in the `matcher`. Similarly, the ComputeDesc specifies the location for the computation's result, using a reference into the `maker` template. For example, the ComputeDesc shown in Figure 4.15 specifies that the first argument comes from the data variable 'newSource' in the database event. The second argument comes from the parameter named *amtPerInch*. The computation identified by 'scaledVal' calculates the number of inches that represents a given number amount. The result of the computation becomes the new height for the rectangle image.

A MessageDesc is similar to a ComputeDesc, but only appears in the response descriptions of DBConnect specs. The `computeName` of a MessageDesc spec must name a message that is understood by the source object represented by the DBConnect. MatchMakers describe the creation of internal events when the `computation` is defined to be an InternalEventType spec. Figure 4.16a is the diagram notation for MatchMakers defining internal-event creation, and an example MatchMaker is shown in Figure 4.16b. If the created event is to carry some information to the Interaction executor that creates it, the InternalEventType spec includes a reference to the appropriate template(s) in the `matcher`. In the example, the event will carry the `sourcePtr` value of the DBConnect that is creating the event. The `maker` template and `map` do not need to be defined for event creation, and thus are omitted from the diagram notation.

A SpawnDesc's elements include an `outlineName` and a `paramVals` array defining any parameter values need for generating the new display. The `spawnSource` defines the new display's source object, using a Path relative to the source object of the containing Outline.

4.4.3 Describing Changes in Display Format

The format of a display includes the arrangement of its subcomponents and the display content other than the source values being displayed. Several kinds of display changes are considered format changes:

- Changes to labels and other "surrounding" display features that convey meta-information rather than object state; examples of these features include labels for attribute names, the numbering in a list, or the notation specific to a type such as Date
- Changes to decorative features that influence the look-and-feel of a display
- Addition or deletion of display components to match the composition of the source object; display components could mean individual interactors or subdisplays that were defined using a Deferment
- Changes in the *focus* of a subdisplay, i.e., changing the path being represented by that subdisplay; in the example display, changing the column of Meal displays is essentially displaying a different path of the Diet object within the same subdisplay
- Changes to the visual representation of object connections, such as those expressed through the combination of a Correspondence and DBRelConnect

In systems that do not support dynamic representation, such changes are made in a location that is consulted only when the display is initially generated and thus the features can change only between invocations of the display or the application that runs it. Some systems that support dynamic representation will consult display descriptions at runtime, but re-generate the entire display to perform the change, thus losing the display state that exists at the time of the format change. Certain aspects of the display are not reproducible from information stored in the database, yet are still relevant to the user's interaction with the display, and thus should be preserved. For example, suppose the user is working with a display in which one subcomponent is a scrollable view of a large text file, and the user changes the display format to add a new subcomponent. The scrolling position for the text display should be preserved so the user does not lose the context in which he or she was working. ODDS provides the ability to describe format changes such that the display designer can define when to preserve relevant display state. display state.

Aside from those described with a Correspondence, format changes are expressed through the combination of a ChoiceMap and either a ParamDesc or FormatDesc spec. A ChoiceMap spec defines the set of alternative formats, while a ParamDesc or FormatDesc spec defines when the change will occur and how to determine which alternative will be generated.

A ChoiceMap includes a mapping from a *handle* to the format alternative identified by that handle. The ChoiceMap spec is positioned within a Layout or Interaction spec at the point where the chosen alternative is to be inserted. The diagram notation for ChoiceMaps is illustrated in Figure 4.17, which shows the entire **MealChoices** Outline. (The display generated from **MealChoices** was pictured in Figure 2.3 and described in Section 4.1.3.) Note that in previous examples (see Figures 4.3.1 and 4.13), diagrams presented a specific format in places where a ChoiceMap spec is shown in Figure 4.17.

A ChoiceMap also has an `initial` element that defines which of the choices is used for the initial generation of the display executors. There are several ways to indicate the initial choice:

- A direct reference to the desired alternative
- The name of a parameter whose value is used as the selection value; using a parameter name to define the initial choice for a ChoiceMap also indicates that future choices made during the display's execution will be dependent on the value of that parameter
- A Path spec defining the path whose value is used as the selection value; a Path is used when the display format depends on a certain condition in the source object, e.g., whether or not the RecipeItem has a 'form' value determines the number of subcomponents in its display

At any given instant in the display execution, the alternative being used depends on a *selection value*, which matches one of the handles in the ChoiceMap and thus specifies what the current format should be.

Separating the description of triggering conditions from the format alternatives allows for several format changes to be triggered at once. Several ChoiceMaps can define a common Outline parameter as the location for the selection value; as a result, updating the parameter's value will trigger multiple format changes. A ParamDesc spec defines actions that update a parameter value. The `paramName` element in a ParamDesc defines the parameter to be updated; the `evalAct` element is a MatchMaker defining an operation whose result value becomes the new value for the named parameter.

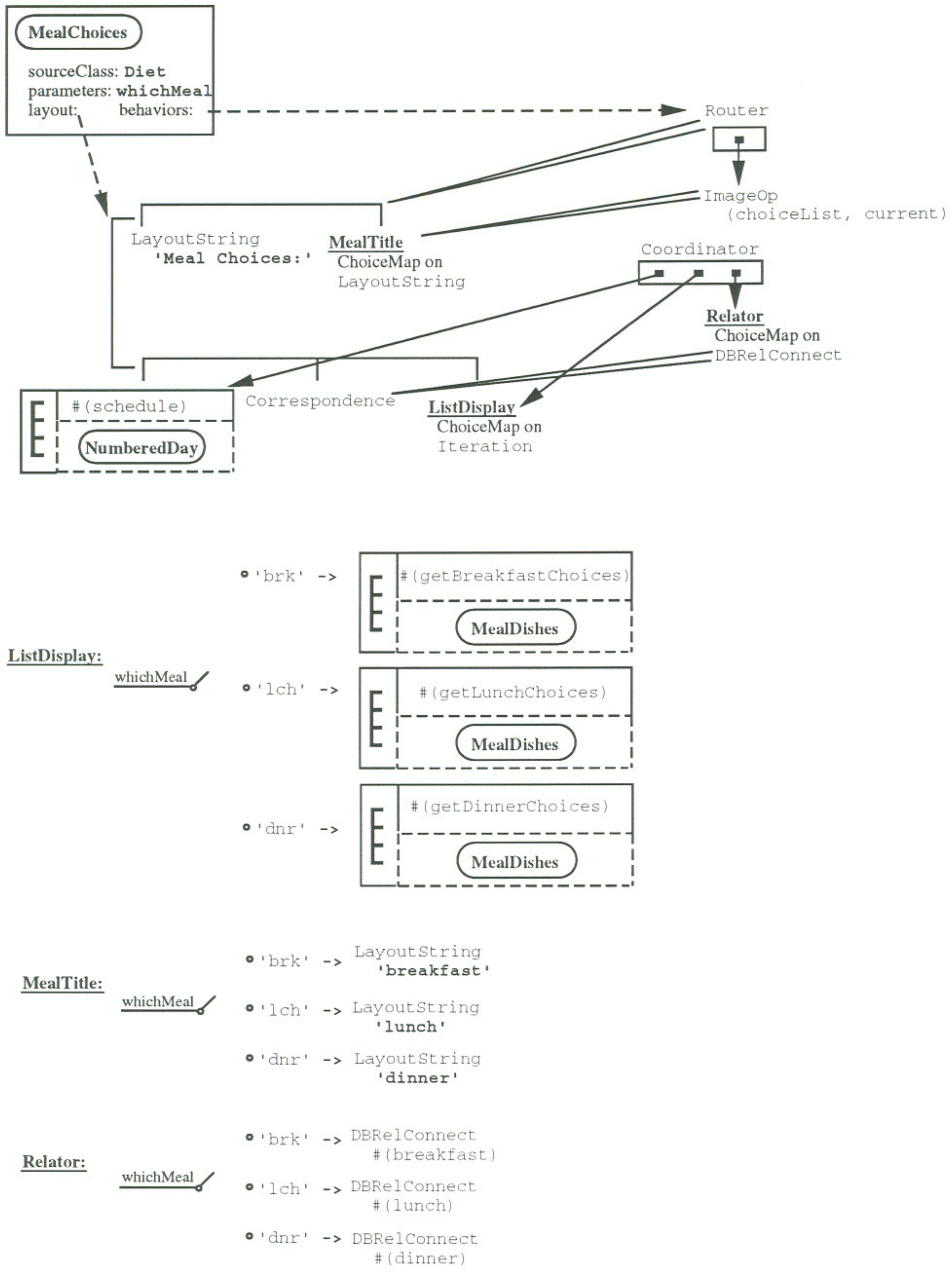


Figure 4.17: MealChoices Outline

A FormatDesc spec defines an action that produces a new selection value and invokes a format change. A FormatDesc is placed in a response description within the Interaction spec for the ChoiceMap. A FormatDesc would be used in situations where the selection value is consulted by just one ChoiceMap and thus does not need to be accessible from anywhere outside the Interaction spec.

The meal-assignment display provides an example where several format changes occur in conjunction. Recall that the display will present a different relationship whenever the user clicks within the title bar. The switch in display focus consists of three distinct format changes: 1) showing the appropriate relationship name in the title bar, 2) changing the list of meals from which to choose, and 3) presenting the connections for the relationship. The ChoiceMaps that describe these changes are all dependent on the Outline parameter *whichMeal*.

The changes made to *whichMeal* are defined in the ImageOp controlling the ChoiceMap labelled MealTitle. To describe this behavior, the response for a *buttonClick* event is a ParamDesc whose `evalAct` defines an operation that updates the value in *current* to be its successor value in the *choiceList* array.

Chapter Summary

This chapter discussed the construction model that underlies the ODDS specification framework, and also described the spec classes making up the framework. The spec classes fall into three general categories. The Layout and VisFeatureSet specs make up one category, and are used to describe the display images. A second category of specs are those used to describe the action sequencing, semantic actions and non-semantic actions that make up display behavior. A third category is the Outline-building specs, which provide the ability to compose or generalize Outlines, thus allowing Outlines to be reused in different contexts. Chapter 5 provides more detail on how these specs support Outline reuse.

The executor classes used in ODDS-Runtime parallel the spec classes that were described in this chapter. When an Outline is invoked, executors are generated to match the object composition in the Outline, and they produce the display images or behavior as defined in the Outline. The generation and operation of executors is discussed in Chapters 6 and 7.

Chapter 5

The External View: Developing Displays and Applications

This chapter explains how the ODDS specification framework is used to design displays and how the display specifications are invoked by an application program. A *complete application* refers to the combination of the displays and the application that creates and interacts with the displays. The development of a complete application involves several roles: the display designer, the application programmer, and the database designer. ODDS is used as a tool in the first two roles, but not for the third. However, since some communication between the displays and the application is channeled through database changes, creating a complete application involves some additions to database classes.

Sections 5.1 and 5.2 discuss the steps involved in display design and in creating an application that interacts with ODDS displays. Section 5.3 discusses the degree of dependence between the three designer roles when using ODDS. The final section summarizes the benefits offered by ODDS for display design and complete application design.

5.1 Tasks for Describing Displays

The display designer's main task is to construct Outlines for the classes of the objects to be displayed. Additional tasks involve ensuring that the procedural components (database methods and computation blocks) that are referenced from an Outline are present in the system. Before discussion of these design tasks, Section 5.1.1 provides a brief introduction on services generally available in OODBMSs.

5.1.1 OODBMS Services

OODB technology extends traditional database services to include the expressive capabilities of a behavioral database model. In particular, the expression of object identity and arbitrary complexity of object structure simplifies modelling of complex data domains. The concept of a class provides user-defined types and the notion of inheritance for sharing type information. The message paradigm supports encapsulation of data and operations on data.

OODBMSs support the behavioral model by providing one or more programming interfaces through which applications access or create database objects and send messages to them. Often there are several means for creating and manipulating objects. Other than sending messages via a data manipulation language, a programming environment may provide visual tools such as forms or inspectors that allow navigation through objects. The OODBMS might also provide a library of base classes that model general constructs such as collections.

To design a database for an application domain, one defines new classes having attributes and relationships that model the structure of application entities. In addition, the classes describe the entities' behavior through its methods. The database designer also creates collections, sometimes called the database extents, that hold the objects making up the database; these collections are assigned to named variables. The database classes and variables reside in a global space where they are accessed by applications. In some systems, the global space is organized into several areas that differ in their accessiblity: some areas hold classes and variables available to all database users, while other areas are available only to a specific group or a single user.

5.1.2 Constructing Outlines

Outlines specs are database objects, thus they can be constructed through any of the object-creation tools provided by the OODBMS. Ideally, a specialized editing tool (built using ODDS) would provide guidance and feedback throughout the construction process. An Outline can be placed in any of the database spaces, depending on who is expected to use it. For example, a database administrator might create Outlines for a basic class (such as String) and make it available to all users. A display designer could create a String Outline that is private to his or her own user space, or place it in a space that is shared with other database users.

The ODDS construction model suggests a general approach for building up a display description. The designer first describes the display image using Layout specs, inserting Path specs to represent data values that are obtained from the source object. Descriptions of input-handling behavior can then be attached to the parts of the image where input is expected. The input-handling is described by a Router and one or more ImageOp specs that are inserted as its `subInteractions`. If the display needs to recognize input sequences that involve several interactors, Coordinators describe how to handle internal events generated by interactor behavior, thus describing the sequencing of activities at the level outside of individual interactors.

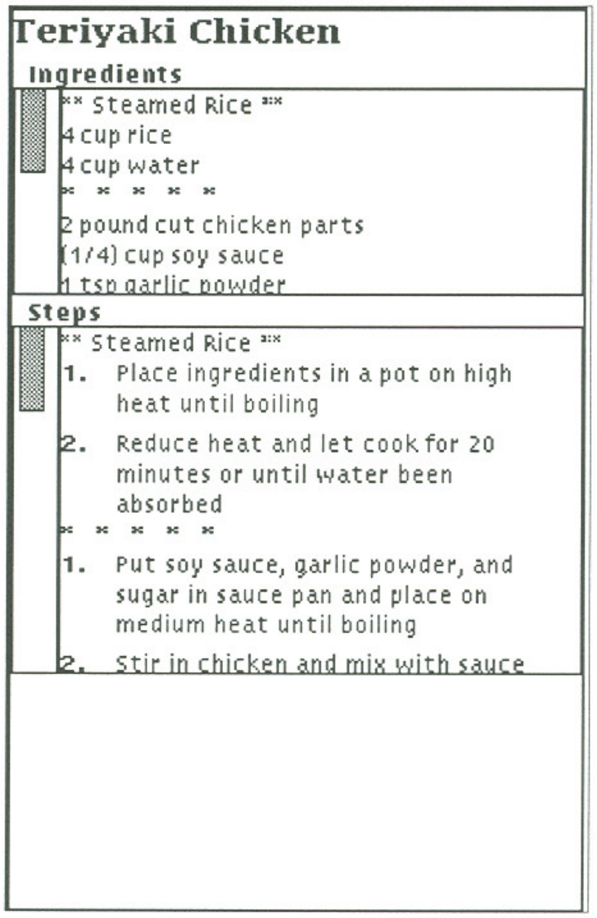
To describe behavior for semantic feedback, the designer identifies the display activities that depend on source-object changes or otherwise require semantic information. DBConnect or DBRelConnect specs are created to define those activities, as well as activities that update source objects. Coordinators define action sequencing among these activities, in addition to those involving input handling.

Composing Outlines through Deferments or Iterations is another approach in which to build up display descriptions. A designer might use Outline composition for various reasons. The most evident reason is to describe an assembly of subdisplays by deferring to the Outlines that describe those subdisplays. For example, the Recipe display in Figure 5.1 can be described as a composition of the displays for a Recipe's ingredients and steps. Furthermore, the display of a RecipeItem (an ingredient) is composed of displays for the RecipeItems "amount", "form:, and "ingredient" values.

Another reason for composition is to reuse an Outline that describes general functionality; i.e., it describes a subdisplay that is not specific to any source class. Examples of a general subdisplay include a scroll bar or scrolling view. In cases where a certain behavior is fairly complex and is required in several places, describing the behavior in a separate Outline and deferring to it saves the designer time and requires less object space.

Outlines that are specific to a given source class might also be created specifically for reuse. The Outline for the Recipe display provides an example. A Recipe sometimes has an ingredient that is also a Recipe; such an ingredient is called a sub-recipe. In Figure 5.1, the display is shown in a mode where the ingredients of any sub-recipes are presented together with the recipe's ingredients. Note that the only difference in displaying a sub-recipe's ingredients is that the ingredient list is preceded by the sub-recipe's title. Thus, the Outline for the ingredients display can be reused within the the Outline for sub-recipe ingredients.

The specification framework supports other techniques for reuse when building Outlines: editing a copy of an existing Outline, reuse through parameterization, and reuse via the class



Teriyaki Chicken

Ingredients

** Steamed Rice **
4 cup rice
4 cup water
* * * * *
2 pound cut chicken parts
(1/4) cup soy sauce
1 tsp garlic powder

Steps

** Steamed Rice **
1. Place ingredients in a pot on high heat until boiling
2. Reduce heat and let cook for 20 minutes or until water been absorbed
* * * * *
1. Put soy sauce, garlic powder, and sugar in sauce pan and place on medium heat until boiling
2. Stir in chicken and mix with sauce

Figure 5.1: Recipe Display with Merged Ingredients and Steps

hierarchy.

Copy and Edit. A new Outline can be constructed by copying an existing Outline or some part of it, then editing the copy to customize the spec for a different context. For instance, an Outline can be made appropriate for a different source class by changing the Path specs within it. Another typical situation is to customize the Interaction specs defining generation of internal events, when the new Outline describes a subdisplay that must interact with other subdisplays in a specific way. For example, in one context a Meal display's behavior might involve only reflecting the current source-object state. In the context of the meal-assignment display, a Meal display has this behavior, but must also respond to a button click by highlighting its image and sending out an internal event.

Generalization of Outlines using Parameters. Generalized Outlines support reuse in the sense that a single Outline can produce multiple variations of a display. A simple example is the **ScrollView** Outline shown in Figure 5.2. The **ScrollView** Outline describes the layout and behavior of a scrollbar and viewport. The *viewWidth* and *viewHeight* parameters enable variation of the viewport size when generating displays. More complex variations can involve colors, source paths within displays, or the positions of certain images in a display's layout. The **NutrientData** Outline discussed in Section 8.1.2 provides an example of parameterizing those features.

If the `subOutline` element of a Deferment is a parameter reference (denoted by the Symbol #PARscrollDisplay), the deferred Outline varies depending on the parameter value. In **ScrollView**, the Deferment labelled <u>contents:</u> defines the subdisplay that will be scrolled, and its `subOutline` is a reference to the *scrollDisplay* parameter. The parameter value supplied during generation determines the name of the deferred Outline used for generating the viewport contents.

Reuse through Class Hierarchy. Outlines may be reused from a class's superclasses. An Outline spec of a class's superclasses may be used to display its instances. Outlines defined for a class `C` are likely to reuse (defer to) an Outlines for `C`'s superclass, using the deferred Outline to describe a subdisplay that presents attributes or relationships inherited from the superclass.

5.1.3 Additional Tasks

After creating an Outline spec, the display designer should ensure that the required database methods have been implemented. The messages used by an Outline are found in Path and

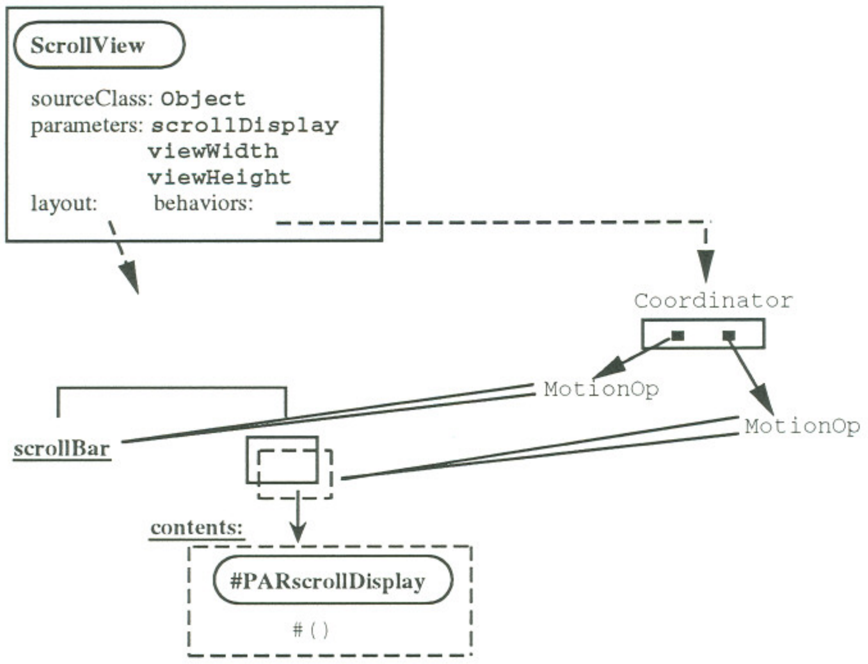


Figure 5.2: **ScrollView** Outline

MessageDescs specs in the Outline. For example, in describing the Recipe display, the message ''subRecipes" must be present in Recipe's protocol since it is named in the source Path of an Iteration spec. In some cases, a display might require queries on source objects that were not anticipated by the database designer. In such a case, the display designer must add a message to an object's protocol. For example, the display of a RecipeItem (an object that represents a Recipe ingredient) checks whether the RecipeItem indicates some form of preparation such as *chopped* onions or *sliced* mushrooms. The message to perform this check, called "hasForm", is needed to choose the proper format for the RecipeItem's display image. The database designer might not provide a message to test whether the RecipeItem's form is a null value (i.e., nil or an empty string), thus the display designer would have to add such a message.

Another general reason for adding a database method is that the display might need to derive a composite object and display it; for example, a list of ingredients sorted in alphabetical order. Sorting the list may be useful only for display reasons, but not for defining the semantics of a Recipe. Thus the operation would probably have to be added to `Recipe` as a display-related method.

In addition to adding database methods, the display designer must write the procedural code represented by the ComputeDesc specs in an Outline, if any. A block of code must be

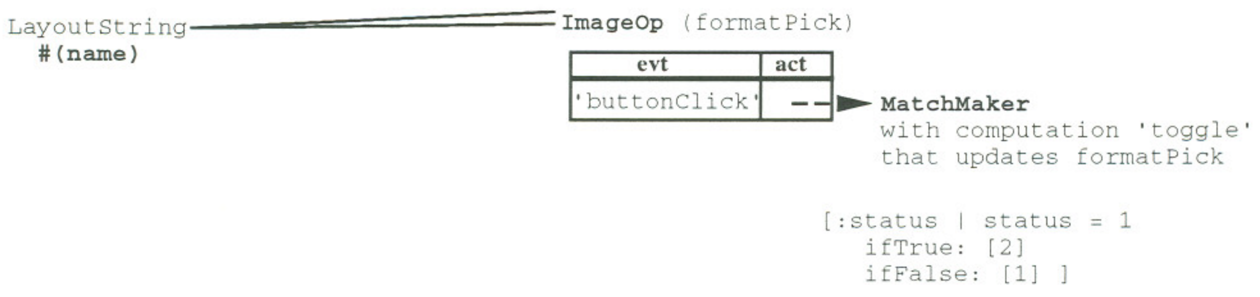


Figure 5.3: Specs for Title in Recipe Display

registered into the Computation Library, using the `computeName` element in the ComputeDesc as the registration key. Figure 5.3 shows the ComputeDesc spec used in **RecipeData**, which describes the computation to toggle between the two modes of the Recipe display (switching between modes occurs when the user clicks on the recipe title).

Finally, another task required in the ODDS prototype is to augment source-class's methods with code that will notify the Source-Update Manager that one or more paths in a source object have a new value. If the value of an instance variable is of interest to an Outline, a notification is placed in methods where that instance variable is updated.[1] If a Path spec names a method that computes a derived value, deciding where to place a notification is not as straightforward, requiring some knowledge of the execution of the source class' methods. The placement of such a notification is discussed further in Section 5.3, which deals with the interaction needed between display design and database design.

The requirement to insert code into database methods works against the separation between display and application definition. An added inconvenience is that notifications will be sent out for any instance of a class, whether or not a notification is relevant tothe displays. Ideally, the DBMS would automatically detect certain kinds of source updates, rather than requiring that database methods include update notifications. The DBMS would then take over the Source-Update Manager's function of gathering update information.

5.2 Application Usage of Outlines

An application program that uses ODDS is not completely free of handling user-interface functionality. The application determines the overall sequencing of its user interface, controlling

[1]This requirement suggests a guideline for the database designer to give each instance variable a single access method that other methods should use instead of updating the instance variable directly.

what displays are initially brought up, then creating and closing displays as necessary throughout its execution. The application program perceives ODDS-Runtime and the DBMS as being one system, since the services from both are accessed through a single interface, the Application Communication Layer. The DBMS services are extended to include creation of displays and control directives for interacting with displays. Thus, the ability to display objects becomes another dimension of the data management provided by the DBMS.

When an application relies on a user-interface tool to implement interactive displays, there are three basic alternatives for the control relationship between the tool and the application [Hayes85]:

- *internal control*, where the application invokes display services somewhat like procedure calls; a display-related service is invoked and completed before the application resumes. User-interface toolkits such as the X Toolkit or the Macintosh Toolbox use the internal control approach.
- *external control*, where the tool controls execution and calls upon application-specific procedures to perform some task and then terminate. UIMSs that model dialogue control as a transition network or context-free grammar use external control.
- *mixed control*, where either the displays or the application program may invoke the other's services, meaning that the control relationship is internal or external at different points during execution.

Mixed control was chosen for ODDS because the displays and the application program each require the services of the other at intermediate points within their threads of execution. Thus, ODDS-Runtime or the application program voluntarily give up control to the other at certain points in its execution.

ODDS-Runtime and the application may be running as concurrent processes, however only one is active at a given time and the other remains suspended; i.e., the components execute as coroutines. Thus, one could view the complete application as having a single thread of execution that runs within the application at certain times and within ODDS-Runtime at others. Using a coroutine approach addresses the need to coordinate access to the database objects from both sides. Whenever control passes from one side to the other, the side receiving control is informed of which objects were updated while the other side was active.

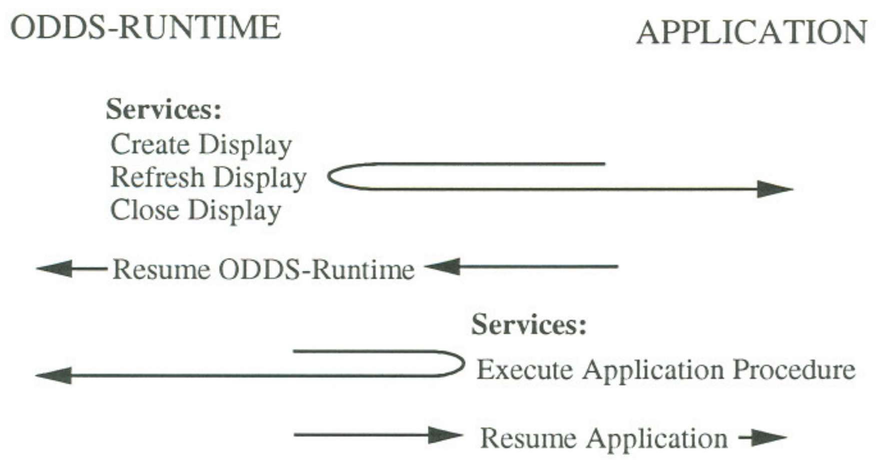


Figure 5.4: Services provided by ODDS-Runtime and the Application

The diagram in Figure 5.4 shows the services that ODDS-Runtime and the application provide each other. When an application is started, it has control and may pass either *temporary* or *full* control to ODDS-Runtime.

When granted temporary control, ODDS-Runtime performs a specific service without processing user input, then returns to the application. The services performed under temporary control include creating, closing or refreshing displays. Some situations where the application would grant temporary control are: 1) to refresh some displays at different points in a computation, to present the intermediate results and 2) to create a notifier display before starting a lengthy computation. The application requests ODDS services by sending messages to an Application Communication Layer. When the application grants full control, it allows ODDS-Runtime to resume the displays; i.e., to accept user input and perform the necessary response actions.

ODDS-Runtime grants temporary control to the application when display execution requires invocation of some computation that is implemented as part of the application. We refer to such computations as *application procedures.* When ODDS-Runtime relinquishes full control, the displays remain unresponsive to input until the application grants full control once again.

An application procedure that is invoked from a display can use the services of ODDS-Runtime that grant temporary control. The procedure is not allowed to resume ODDS-Runtime in the middle of its execution, because a procedure is viewed as a unit of computation that completes its task before any other computation takes place. If the procedure were to resume the displays, the user may cause invocation of another procedure, and the two procedures could interfere with each other's work if they operate on common database objects.

Since granting temporary control is conceptually similar to invoking a procedure, the application and ODDS-Runtime are in an internal-control relationship when the application has full control, and in an external-control relationship when ODDS-Runtime has full control. All of the control shifts described above involve supplying the newly activated component with information on updated source objects. Thus, when regaining control, the application can examine the database changes made by the display system and respond to them if necessary.

The following is a scenario illustrating the control exchange between ODDS-Runtime and an application that uses the Recipe display:

1. The application creates the Recipe display and a command panel that includes an area for displaying feedback messages. Assume the Recipe display has an additional command button that recalculates the Recipe's ingredients to produce a different number of servings. The Recipe display also includes a text field for entering the desired number of servings.

2. The application grants full control to ODDS-Runtime. User enters the desired number of servings and clicks on the Recalculate button.

3. The Recipe display invokes an application procedure to perform the calculations, thus granting temporary control to the application.

4. The procedure updates the Recipe, printing out to the message area in the command panel to inform the user of the sub-recipes that have been processed. To display each message, the procedure grants temporary control to ODDS-Runtime.

5. When ODDS-Runtime resumes full control, the user goes to the command panel and clicks on a button, invoking a command to switch to a different screen of displays. The panel display responds by setting the appropriate database variable as a signal to the application that the user has requested that command. The panel display then returns full control to the application.

6. The application detects that the database variable has changed. To execute the command, the application makes several requests to ODDS-Runtime, granting temporary control as necessary to close displays and open new ones.

The ODDS prototype currently does not implement a mechanism for generated displays to use application procedures. Adding the mechanism entails some small additions to the specification framework and to the Application Communication Layer. In the specification framework, a new kind of spec must be added to represent use of an application procedure. The spec would have similar content and be used similarly to a ComputeDesc. To keep the Outline from being tied to a particular application, the Outline could be generalized with a parameter that holds the name of the application procedure. The application must register the procedures required by the Outlines that it invokes, identifying each procedure by the name defined in the Outline.

The application may need to display objects that are not persistently stored in the database. To use ODDS' services in this case, the application can create *transient* objects, which exist within the same space as persistent objects, but are not saved as part of the database. Thus, these transient objects do not exist past the execution of the application that created them. While the application runs, a transient object is accessible to ODDS-Runtime and can be a source object for a display.[2]

To summarize, the basic tasks of an application are to:

- Perform the initializations necessary to use ODDS' services. The application must connect to ODDS-Runtime through the Application Communication Layer, register any procedures required by the Outlines it invokes, and set up any global variables needed to communicate with the displays.
- Create displays and manage exchange of control with ODDS-Runtime as necessary.
- Create a handler procedure that examines updated source objects for relevant changes when temporary or full control is regained.

5.3 Interaction among Design Areas: Display, Database, and Application

As noted earlier, producing semantic feedback typically results in a tight coupling between application and display specification. With ODDS' architecture, the coupling is reduced by treating display capabilities as part of the data definition rather than a function that the application performs. The application program does not deal with the internal workings of its

[2]To distinguish between persistent and transient objects, some OODBMSs designate a persistent root object and all objects reachable from the root are considered persistent. An alternate approach is to have the programmer explicitly mark objects as being persistent or transient.

displays, and mainly controls a display as a whole using a fixed set of requests to the runtime system. However, the descriptions produced by the display, database and application designers rely on each other to a certain degree. This section discusses the overlap among the roles and how changes made to descriptions in one area can affect the descriptions in another area.

As noted in Section 5.1.3, the display designer needs to make certain additions to database methods that define the source objects' behavior. One kind of addition is notification to the Source-Update Manager about changes in a source object's state. To ensure that notifications are sent at the proper times, the display designer may need to be aware of the execution dependencies built into the behavior of the source objects. This knowledge is needed when the display designer wishes to present a path value whose state is derived from other objects; e.g., the #(nutrients) value for a DayPlan is derived from nutritional information in its breakfast, lunch, and dinner Meals. (The objects that the derived value depends on are called *participants.*) If the designer expects the display to be notified of changes in a derived value, he or she must ensure that the notification is executed whenever any participant is updated. The defined source-object behavior may already arrange for the deriving method to be executed whenever a participant changes. If not, the display designer should consider that this execution dependency is a requirement of the display rather than the source object semantics, and instead define the dependency within the Outline.

Another task performed by the display designer is to create any methods needed to support an Outline. Allowing the display designer to add database methods raises a question of whether the modularity between definition of displays and source-object semantics is compromised. An Outline uses database methods for several reasons: 1) to obtain a value along a certain path in the source object, 2) to check for certain conditions in source objects that trigger format changes or other display responses, 3) to obtain a result from an algorithm or computation over database objects and 4) to update source objects. The first two uses only read values in source objects, so methods for those purposes can be added without violating existing integrity constraints or requiring changes in existing methods. The third and fourth uses may update values and may violate constraints, unless certain precautions are taken.

To ensure integrity, methods added for display purposes should not update instance variables directly, and should only use the original database methods that define source-object behavior. If these restrictions are followed, methods can be added for support of displays without risk of affecting the semantics defined by the database designer. Thus, once the public protocol for a source class is stable, display-related methods can be implemented fairly independently of existing database methods. To use existing methods in creating new ones, the display designer requires only an understanding of what they accomplish, not the implementation within them.

Certain situations require involvement between display and database designers. If the database designer wishes to delete or modify the functionality of a database method (one created by the database designer), then Outlines or display-related methods referencing the changed method become invalid, requiring maintenance by the display designer. Some aids are available for finding the Outlines affected by an altered or deleted method. Outlines can be queried about the database messages they invoke. The designer must search through the Paths and MessageDesc specs in the Outline, which constitute the interface to source classes. Changes made by the database designer might also affect the execution dependencies in the source object behavior, and therefore update notifications that were placed by the display designer may require maintenance as well.

The addition of subclasses to the database schema is another situation that affects the display designer. The instances of a new class can be displayed using the superclass' Outlines, but typically require additional display features to present their specialized structure and semantics. The display designer also needs to examine Outlines that might present the new class's instances in a subdisplay. If an Outline is intended to make use of the Deferment's dynamic binding capability (i.e., produce different subdisplays according to the class of the subdisplay's source object), the display designer must ensure that the new subclass has a specialized Outline with the name used in the Deferment.

The interaction between application and display design is not as involved as the interaction between display and database design.[3] An application would be affected if an Outline that is used is modified and describes a display that no longer suits the application's needs. This situation can be addressed simply by finding (or creating) a more suitable Outline and changing the application program to use that Outline's name. The application is affected also when an Outline is modified to include invocation of an application procedure as part of the display behavior, or the procedure name in an existing invocation is changed. The application programmer must ensure that the appropriate procedure is registered (with the Application Communication Layer) under the new procedure name.

[3]Application design and database design also have interdependencies, however they are not directly relate to usage of ODDS.

5.4 Advantages

The ODDS specification framework permits the display designer to specify the fine details of a display's appearance and behavior, yet maintains a level of abstraction that detaches the designer from concern of how the specified activities are implemented. Thus, the designer can focus more on concerns specifically related to display activities rather than general programming tasks. Working at a higher level of abstraction also simplifies modifications so that a specification may be developed incrementally and reused in different contexts more easily.

In particular, when describing spatial composition of the display image, the designer works in terms of combining subparts using a composition operator, as opposed to calculating coordinates or windowing transformations to place the image subparts properly on the display screen. When describing possible display formats, the designer can indicate reuse between formats by having the formats refer to a common spec, and not be concerned about the way in which shared executors are detected and saved for reuse.

When describing display behavior, the specification constructs abstract away implementation concerns that fall into the areas of input handling, event management, display updates, and communication with the database. Declarative description of the coordination among units of display behavior is an important abstraction contributing to the description of complex display responses. In several object-oriented frameworks, coordination of behavior is often accomplished through a dependency mechanism. This approach makes the coordination difficult to see and alter, especially if many objects and dependencies are involved.

In comparison with display definition through programming, the specification framework provides a clearer understanding of where particular display functionalities are defined since it allocates design features to particular kinds of specs. A class library or application framework may provide some guidance through the organization of the class hierarchy, but the methods of a class generally have no organization to indicate where visual attributes or action sequencing are defined. Certain methods may be known to carry out certain functions, such as a `display` method, but in many cases, the instance variables that control the display operation are set in other methods that are executed prior to executing `display`. Thus, to customize a single feature within an existing user-interface component, a programmer may need to understand its entire implementation to find out where the feature can be modified. In some cases, the implementation is spread across a hierarchy of classes, causing more overhead for the programmer.

Finally, ODDS provides the advantage of modularity when designing the display, database, and application portions of a complete application. A clear distinction between display definition and the application program provides a simple and direct approach for modifying what displays are used by the application and for sharing display descriptions among several applications.

Chapter 6

Runtime Activities

This chapter describes the activities of ODDS-Runtime during display execution. This description provides a general picture of the sequence of activities in the runtime system, whereas the next chapter discusses the system's implementation in terms of the responsibilities and data structures managed by each system component. Section 6.1 details the steps of control exchange between the ODDS-Runtime and the application program. Section 6.2 goes through sequences of activities involved in creating displays and carrying out their execution.

6.1 Control Exchange

ODDS-Runtime is implemented as a Smalltalk application accessing an OODB where the Outline specs and source objects reside. The interchange of control between ODDS-Runtime and the application involves several Smalltalk *threads* and a database-session process. A Smalltalk thread represents a series of actions that is executed independently of those executed for any other Smalltalk thread.[1]

Figure 6.1 shows the threads and processes that make up the system execution, indicating which system components perform their work as part of a particular thread or process. The function for each thread or process is as follows:

- The *display thread* carries out the work of the Interaction Manager, accepting user inputs that activate the Interaction executors, which in turn drive display changes. At certain points, the Interaction Manager invokes services of the Display Generator, which are also executed within the display thread. (In the implementation, each display runs in a

[1] A thread is considered a process in the context of a Smalltalk environment, but the term 'process' is reserved to indicate an operating-system (OS) process. The Smalltalk environment runs as a single OS process.

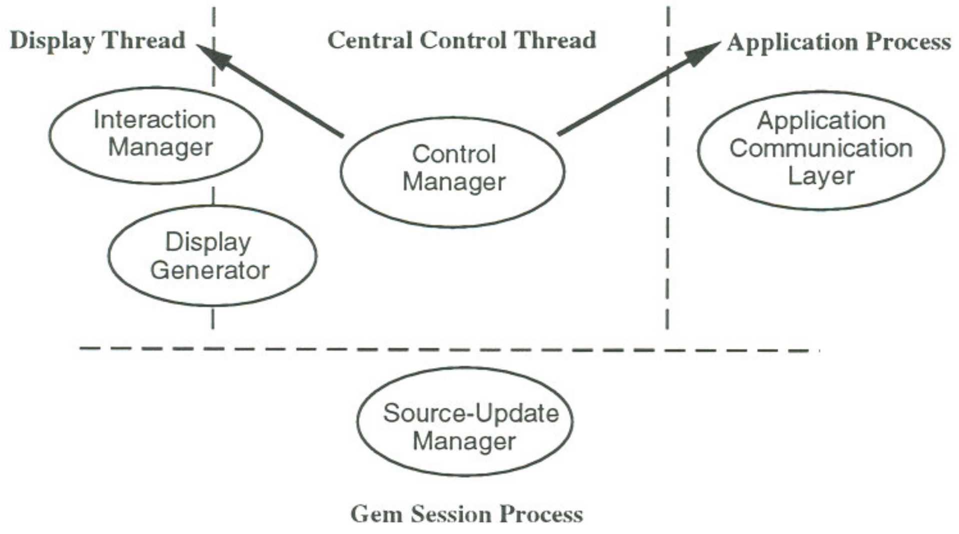


Figure 6.1: System Components and Processes

different thread, but the threads are coordinated so that only one runs at a given time; thus they may be viewed as a single display thread.)

- An *application thread* or *process* is the execution of the application using ODDS. The application may run as a Smalltalk thread within the same image as the display thread or in a remote image. The application could also run as a process that uses an appropriate interprocess-communication mechanism to connect to ODDS-Runtime.

- A *central control thread* performs the work of the Control Manager; i.e., to facilitate service requests from an application to ODDS-Runtime and vice versa. At times, the work of the Interaction Manager and the Display Generator are executed within the central control thread. The Control Manager enforces restrictions regarding the requests that the application or ODDS-Runtime can make at a given time. The service requests and control activities are described in more detail below.

- A database session process is involved because the work of the Source-Update Manager is accomplished within the database. The Source-Update Manager is queried by the other system components through the access mechanisms available in the OODBMS's application-programming interface (API).[2]

[2]Some OODBMSs provide the ability to run the database session in the same process as application using it. The existence of the session process is specific to the ODDS prototype.

The activity of the central control thread basically consists of accepting requests either from the application (through its Application Communication Layer) or from the Interaction Manager. As described in Section 5.2, the application program has a model of passing either temporary or full control to and from the display system. Each service request results in one or more shifts between threads or processes.

When the application requires a service of ODDS-Runtime, the Application Communication Layer sends the request to the Control Manager, thus activating the central control thread. The application process then suspends. To process requests that grant temporary control to the display system, the Control Manager invokes the appropriate component: requests to refresh or close displays are handled by the Interaction Manager, and those to create displays are handled by the Display Generator. During the display creation or refresh, the Control Manager accepts and processes any requests to invoke application procedures, but rejects requests to return full control to the application. When displays are operating under temporary control, granting full control to the application can result in an improper control state in which the application does not continue execution at the point where it had requested the display service. When the application's request has been serviced, the Control Manager notifies the application to continue and the central control thread suspends, waiting for the next application request.

When the application wishes to resume the displays; i.e., grant full control to ODDS-Runtime, the Control Manager notifies the Interaction Manager, which initiates a new display thread. The central control thread suspends, awaiting requests from the Interaction Manager. At this point, the Control Manager can accept requests to return either temporary or full control to the application. Thus, it must distinguish between this point in execution and the one described above, where it awaits request after granting temporary control to ODDS-Runtime.

When the Interaction Manager or Display Generator requires service from the application, the component submits the request to the Control Manager and the display thread suspends, thus reactivating the central control thread. A request to execute an application procedure grants temporary control to the application. The application process becomes active and the Application Communication Layer invokes the appropriate procedure. When the application has temporary control, the Control Manager disallows any application requests to grant full control to the displays. When the application procedure has completed execution, the central control thread then activates and in turn activates the display thread, thus completing the requested service.

When a request is made to return full control to the application, the application process is notified, and the central control thread suspends, waiting for requests from the application.

6.2 Display Management Activities

This section discusses how ODDS-Runtime carries out display generation and execution.

6.2.1 Generation of Executors

When the application requests the initial generation of a display, it identifies a source object and an Outline for the Display Generator. Recall that LayoutExec and InteractionExec classes are Smalltalk classes that correspond to the spec classes described in Chapter 4. The generation procedure first initializes LayoutExec and InteractionExec instances based on data from an Outline spec, then adds the runtime data needed to execute the display. These two phases are called *replication* and *construction*.

In the replication phase, information from the named Outline spec is merged with the source object's path values for those paths defined within the Outline spec. The combined information is used to create the set of LayoutExec and InteractionExec objects that make up the display. A replication mechanism provided by the OODBMS supports the merging procedure; the mechanism's function is to create a close approximation of a database object in an object space that is external to the database. The replication mechanism supports display generation by handling the duplication of connections between objects, so that the composition of executors matches that of the corresponding specs.

The main steps in the replication phase are as follows:

1. As the Deferments, Iterations, and Paths within the Layout and Interaction specs are being replicated, their corresponding executors are collected and recorded.

2. The values for all the Paths in an Outline are retrieved through a single call to the Source-Update Manager. The Source-Update Manager records the path information in its tables, since these are exactly the paths that must be monitored for changes during display execution.

3. Any Iterations in the Outline are expanded into a collection of Deferments, one for each element of source collection being displayed, which was retrieved as one of the values in the previous step. The newly created Deferments are added to those collected in step 1.

4. The replication steps 1-4 are repeated each Deferment that was collected from the Outline. If the source object's class has no Outline with the name defined in the Deferment, then its superclasses are checked. The Display Generator also informs the Source-Update Manager that the source object and chosen Outline were used to generate a display; the Source-Update Manager uses this information to track the displays' interest in source objects. After generating executors for the chosen Outline, the Display Generator records an association between the Deferment and the generated subdisplay, for use in the construction phase.

In the construction phase, the Display Generator pieces together the subdisplays to form the entire top-level display; the end-result is a single hierarchy of Layout executors and a single collection of Interaction executors. Starting with the executors for the top-level display, the Display Generator replaces all the Deferments with either the Layout or Interaction executors that were generated for each Deferment. Where a Deferment is referenced by a Layout executor, it is replaced with the Layout executor associated with that Deferment. Similarly, where a Deferment is referenced as an Interaction executor, the appropriate Interaction executor is inserted.

Additional data structures that are needed for runtime operation are created based on the Outline's information and are placed in the appropriate executors. Lastly, the Display Generator instructs the Layout executors to render the display image and sets up the Interaction executors to enable them to receive input events.

6.2.2 Display Execution

The activities in display execution basically consist of recognizing and responding to events. Events are generated as a result of user input, database changes, or activities in the Interaction executors. Events cause the activation of Interaction executors, which in turn execute actions according to information in their event mappings.

Event Processing

The Interaction Manager relies on the underlying window system to obtain low-level events from the keyboard and mouse. These events are used to create more abstract forms of event objects that are used to activate Interaction executors. Router executors initially receive all user input events. The activated Router will route the event to one of its `subInteractions`, depending on the event's type. If none of the `subInteractions` are expecting an event of that type, the event is discarded.

The activation of any kind of Interaction executor besides a Router results in the following steps:

1. The `eventMap` of the activated executor is consulted to find the actions that should be executed for the type of the activating event.
2. The list of actions may indicate that certain flags need to be checked. If the named flags are not set, then no further actions are executed and the event is discarded.
3. If the named flags are found to be set, then they are cleared, and each action in the response description is carried out. Any flags specified in the response description are raised.

MatchMaker Execution

MatchMaker executors provide a general mechanism through which actions are executed, therefore I first describe how they function. To carry out an operation, a MatchMaker executor first matches its `matcher` template against the prescribed target objects, and thus identifies the operation's arguments and the objects to be updated. In the MatchMaker in Figure 6.2, the `matcher` holds an ImageOp template that will be matched against ImageOp executor shown. The matching process identifies the two arguments for the `computation` as being an array containing the strings 'breakfast', 'lunch', and 'dinner' and a string 'lunch'. The `map` consists of entries for the tags IM and NXT, and the `maker` template indicates that the target object matched to IM, i.e., the ImageOp executor, will be updated. Futhermore, placement of the NXT tag indicates that the slot named "current" will be set to the result of the computation.

The following steps take place when a MatchMaker carries out its operation:

1. The `matcher` template and its subcomponent templates are matched against either the Interaction executor, its Layout executor, or the activating event, depending on the template's class-type. A template matches a concrete object if two conditions are true: 1) the target object's class is the same as or a subclass of the template's class-type and 2) the template's components, if any, match the values in the concrete object's instance variables. The matching process produces a *matchTuple* dictionary that associates each

```
MatchMaker:

    Matcher
    IM: ImageOp (choiceList -> Object,
                 current -> Object )

    Computation
    ComputeAction (  selector  -> 'nextInLine'
                     arguments -> Array (1 -> IM @ choiceList, 2 -> IM @ current),
                     result -> NXT: Object  )

    Maker
    IM: ImageOp (current -> NXT: Object)


Target objects:
```

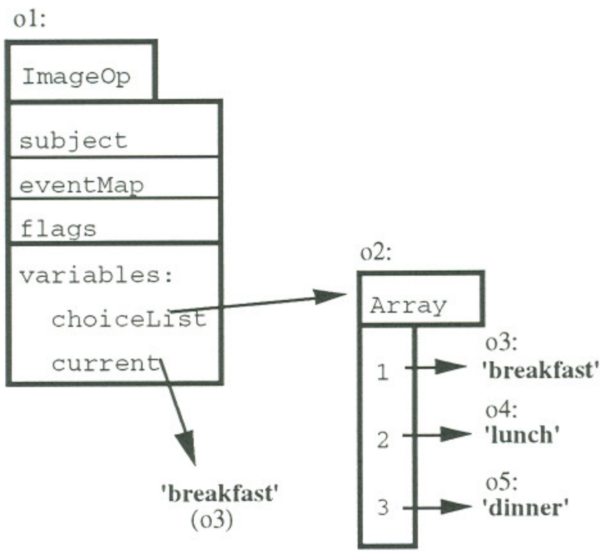


Figure 6.2: MatchMaker Example

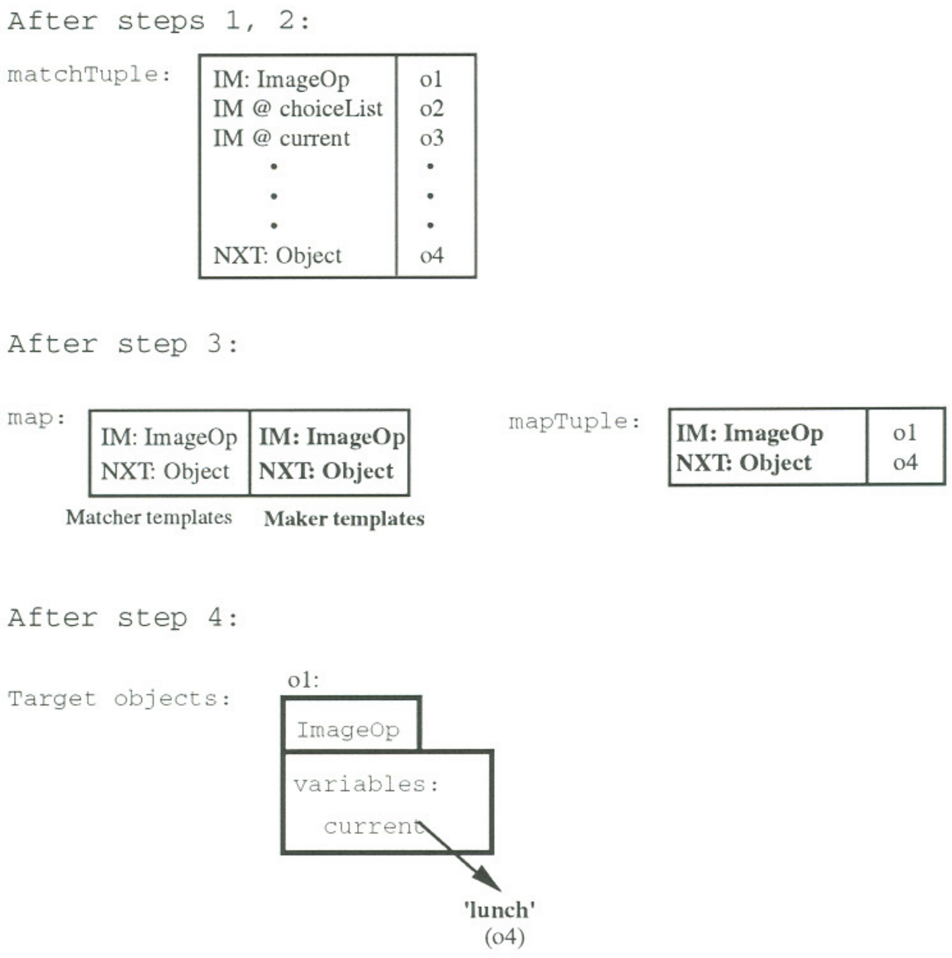


Figure 6.3: Steps in MatchMaker execution

template object to its *concrete match* within the target. Figure 6.3 shows the mapTuple for the MatchMaker example. The first three entries in the dictionary are present at the end of this step.

2. Executing the `computation` element of a MatchMaker results in either a computation, a message being sent to a database object, or the creation of an internal event. The `computation` element may refer to some part of the `matcher` templates, indicating where to obtain either an argument value for a computation or message or a data value that goes into an internal event. When executing the `computation`, an argument or data value is obtained from the matchTuple, using the appropriate template as a key. If an internal event is created, it is forwarded by the Interaction executor that invoked the MatchMaker. For a computation or message send, the resulting value is generally used as a value in an updated object, which is the case in this example. As shown in Figure 6.3, the result value `o4` becomes the concrete match for the NXT template and is added to the matchTuple to be used in subsequent steps.

3. This step and the next occur for operations that update a Layout or Interaction executor. A *mapTuple* dictionary is built up based on the bindings in the `map` (recall that a binding is represented by a common tag in the `matcher` and `maker`). For each `map` binding, the *mapTuple* records the association between the `maker` template and the `matcher` template's concrete match. Thus the `mapTuple` represents positions that will be updated in the target object and the new values for those positions. Figure 6.3 illustrates the `map` and `mapTuple` for the example MatchMaker.

4. Finally, the target object is updated. The `maker` template is traversed, and each sub-template is checked to see if it is a key in the `mapTuple`. If so, the sub-template's corresponding position in the target object is updated to the associated `mapTuple` value. In the example, the `mapTuple` identifies `o1` (the ImageOp executor) as the updated object, and identifies `o4` as the value for the ImageOp variable, 'current', which was marked with the object tag, NXT.

Example: Activities in Meal-Assignment Display

This section discusses some activities occurring in the execution of the meal-assignment display. The types of activities illustrated are feedback to user input, modification to the underlying source objects, and display response to source changes and format changes.

In the display, one example of feedback to user input is that whenever the user moves the cursor within a DayPlan subdisplay, it is highlighted by reversing its foreground and background colors. The MatchMaker spec in Section 4.3.2 is a description this operation. In general, MatchMaker executors affect the display image by updating a Layout executor and then instructing it to redraw its part of the image.

The ImageOp executors for the DayPlan and Meal subdisplays are all `subInteractions` of the Coordinator. In addition, a DBRelConnect executor that controls the display of arrows (through a Correspondence) takes part in the coordinated behaviors. When a DayPlan ImageOp receives a 'buttonClick' event, the responding action is the creation of an internal event that has the `typename` 'daySelection' and identifies the ImageOp's source object; i.e, a DayPlan object . Similarly, an internal event with the `typename` 'mealSelection' and identifying a Meal object is created whenever a Meal ImageOp is activated with a 'buttonClick' event. The Coordinator is activated whenever any of its `subInteractions` generates an internal event. In this example, the Coordinator's `eventMap` dictates that it will forward the internal events to the DBRelConnect, while ensuring that a 'mealSelection' event is forwarded after each 'daySelection' event.

The assignment of a Meal to a particular DayPlan is accomplished through the execution of a MessageDesc, which causes a database message to be sent. In this example, the DBRelConnect executor executes a MessageDesc in response to a 'mealSelection' event. The MessageDesc indicates that the relationship named in the DBRelConnect determines what database message is sent; depending on which format is installed, the message updates either the breakfast, lunch, or dinner connection in the selected DayPlan.

As the database message is processed, relevant changes to source objects are recorded by the Source-Update Manager. Afterwards, the Control Manager returns control to the Interaction Manager, along with an indication of whether any changes have been recorded. If any database changes were made, the Interaction Manager obtains the report from the Source-Update Manager, then refreshes the displays affected by those changes. A database event is produced for each database update and activates the DBConnect or DBRelConnect executor that controls the presentation (Layout executor) for the appropriate part of the updated source object. The details concerning the source-update report and the selection of an appropriate Interaction executor are discussed in Chapter 7.

In the example, the Source-Update Manager reports that the path #(breakfast) was updated within a DayPlan source object; the Interaction Manager then activates the DBRelConnect executor. To reflect the existence of a new relation, the DBRelConnect must determine the pair of Layout executors that represent the domain and range objects of the updated connection. The Correspondence keeps a dictionary of the Layout executors for each domain object. The DBRelConnect determines the appropriate Layout executor for the range object by consulting one of the system tables, as explained in Section 7.3. The Correspondence executor updates the display to reflect the new relation. A LayoutLine for the new Layout pair is created and displayed.

The next functionality discussed is the ability to shift focus between the schedule's breakfasts, lunches, or dinners, triggerred by a button click in the title bar. The format changes involved are coordinated through their dependence on the Outline parameter, `whichMeal`. The 'buttonClick' event is received by the ImageOp for the titlebar, and causes the execution of a ParamDesc that updates the parameter, setting it successively to the strings 'breakfast', 'lunch', and 'dinner'. The MatchMaker executor described earlier in this section performs this operation.

The Interaction Manager goes through the list of ChoiceMaps that are dependent on `whichMeal`, instructing each to produce a new format choice based on the parameter's value. During display execution, the choices within a ChoiceMap executor are in the form of database-object identifiers that represent spec objects, which may be either Layout, Interaction, Path or Deferment specs. At any given time, only one of the choices, i.e., the current choice, has a set of executors generated for it.

Whenever a new choice is selected, the Display Generator is invoked to produce new executors from the spec object. The generation is similar to initial generation, having a replication and construction phase. The main difference in the format's replication phase is that the Display Generator reuses the executors for any specs that were part of the previous format choice. Another difference is the steps the Display Generator must take to maintain system tables. During the format's construction phase, the state of the newly created executor replaces that of the previous format choice and, as in the initial construction phase, various runtime objects (Views, Controllers) needed for execution are added to the new executors.

This chapter discussed the ODDS-Runtime in terms of the sequence of activities involved for interacting with the application, for generating displays, and for managing display operation.

The following chapter provides details on how these activities are implemented, organized by the responsiblities of each system component.

Chapter 7

ODDS-Runtime System Components

This chapter explains how the system components of ODDS-Runtime interact, with emphasis on how semantic feedback is supported in display execution. The ODDS prototype was implemented using UTek Smalltalk and version 2.0 of the GemStone OODBMS. GemStone resided on a Sun 4/75 workstation while the Smalltalk environment executed on a Tektronix 4317 workstation running the UTek 3.1 operating system; the two systems interacted over a local-area network.

ODDS-Runtime employs the GemStone-Smalltalk Interface (GSI), a group of Smalltalk classes that implements interaction with a GemStone database-session process, called a *Gem*. GSI includes general database facilities such as logging in and out of a session and handling client-server communication. In addition, it supports the object-oriented database features provided by the Gem. GSI includes the concept of a *proxy* that is created within the application's data space to represent a database object. A proxy provides a convenient means to send messages to a database object or refer to it for some other purpose. In addition, GSI includes an object replication mechanism between GemStone and Smalltalk objects, which was mentioned in Section 6.2.1.

The system components are implemented as objects: the Control Manager, Interaction Manager, Display Generator, and Application Communication Layer are Smalltalk objects; the Source-Update Manager is a GemStone object. Figure 7.1 summarizes the data or requests communicated between the system components; each arrow originating from a component is labelled with an action performed by the component. The Display Generator creates the executors for the displays when an Outline is invoked and when displays take on a new format. It also records the source objects that are of interest to each executor. The Interaction Manager carries out two general functions to manage the generated displays. First, it processes user input and initiates the proper responses, which may result in display changes or requests

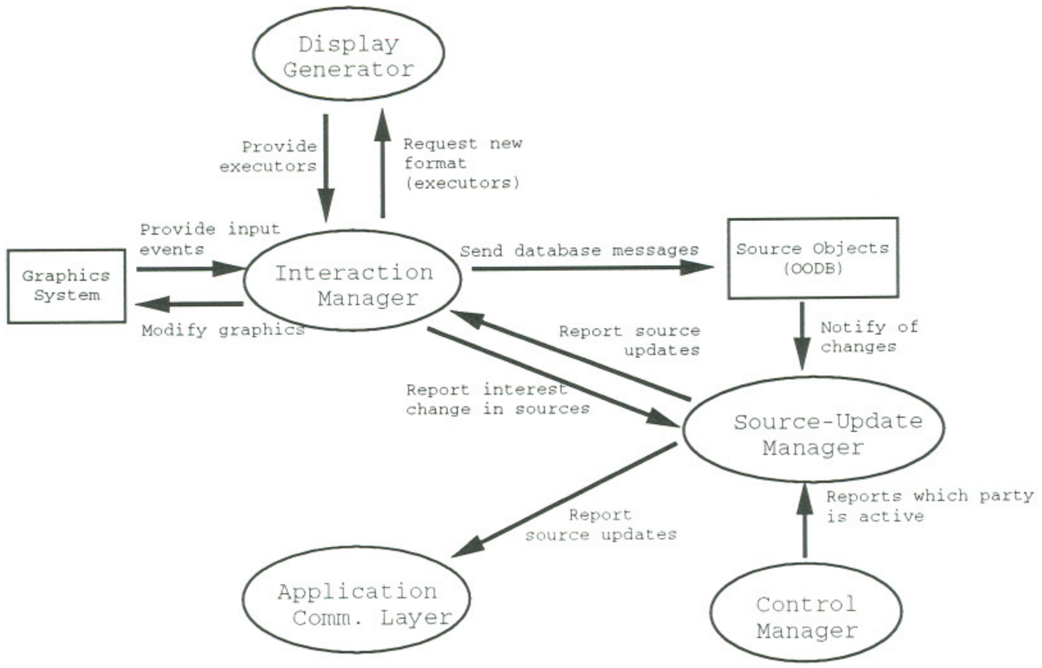


Figure 7.1: Interaction among System Components

for database operations. Second, it responds to database changes, initiating display activities that will present the current state of the source objects. The Interaction Manager obtains information on source-object changes whenever it regains control (temporary or full) from the application or after a database message has been executed. Under certain conditions, changes in the source objects can cause a subdisplay to present a different source object or take interest in different source paths. The Interaction Manager is responsible for tracking such changes for its own uses and also provides this information to the Source-Update Manager.

During the execution of database messages, the Source-Update Manager creates a report of the source-object changes. The report consists of the set of changed objects, the paths changed within each object, and the new value for each path. The report creation is dependent on whether the executing database message was invoked from ODDS-Runtime or from the application; thus the Source-Update Manager relies on the Control Manager for this information. Control exchange between ODDS-Runtime and the application is managed by the Control Manager and Application Communication Layer.

System Tables

Two main system tables are used by ODDS-Runtime in support of executing complex display responses and format changes. First, the *SourceInterest Table* keeps track of all database objects whose state affects the existing displays. This table resides in a GemStone database and is managed by the Source-Update Manager. The Source-Update Manager consults the SourceInterest Table whenever it receives notification of a database change. By checking the change against the SourceInterest Table, the Source-Update Manager distinguishes the database changes that need to be reported to the Interaction Manager. Second, the *ExecNotification Table* keeps track of the executor or executors responsible for reflecting the state of a given source object. This table is a Smalltalk object that is managed by the Interaction Manager accessible to the Display Generator. When responding to the database changes reported from the Source-Update Manager, the Interaction Manager consults the ExecNotification Table and activates the appropriate executors for handling each change.

To serve the purposes described above, certain invariants must be maintained in each table. First, both tables must keep a list of entries for all the source objects currently being displayed. The set of displayed objects changes due to updates in source-object composition or changes in display focus. Another point at which the set is updated is when a display is closed. Second, each entry must keep a current record of the paths being displayed for each source object. The set of displayed paths might be changed when a new format is chosen.

The SourceInterest Table has an additional requirement to hold entries for any non-displayed database objects whose state can affect existing displays. In particular, objects at an intermediate position within a multi-step path must be monitored. Consider a path #('step1' 'step2' 'step3') that has the object `root` as its starting object. The intermediate objects in this path are `o1` and `o2`, which are the values for instance variables `step1` and `step2`, respectively. If the value for either `step2` or `step3` is updated, then the end-value of the path, `o3` will (most likely) also be changed. Thus interest must be registered for `o1` and `o2`.

At this point, the ExecNotification Table is described in more detail. (The SourceInterest Table will be described in connection with the Source-Update Manager, in Section 7.3.) At all times the ExecNotification Table keeps a record of the database objects being displayed and the display(s) representing each object. In addition, the ExecNotification maintains a list of Interaction executors interested in a given path within the source object. Some terminology is needed to explain the contents of the table.

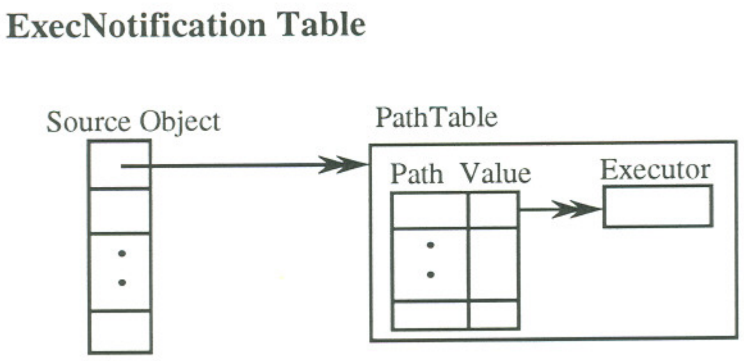


Figure 7.2: Structure of the ExecNotification Table

A *display unit* is a display or subdisplay generated by an Outline invocation. The screen image and behaviors of a display unit are produced by its set of Layout and Interaction executors. A display unit is an external representation for a source object, presenting the values of certain paths within that object. For example, the paths #(ingredients) and #(instructions) are presented in the Recipe display (Figure 5.1). A display unit may contain other display units, brought about by including a Deferment spec within an Outline. A display unit created as a result of an application request is called a *top-level display*, while one created due to a Deferment spec is a *subdisplay*.

In the ExecNotification table, a display unit is represented by a data structure called a *PathTable*, which holds a mapping from each source path presented by the display unit to the executors interested in that path. An interested executor will be either a DBConnect, DBRelConnect, or Deferment executor. The latter case indicates that the path value is being presented as another display unit, i.e., as a subdisplay.

The structure of the ExecNotification table is motivated by some functional requirements for the displays. One is that multiple displays of a source object should remain consistent with each other. Thus, the ExecNotification table maps each source object to one or more display units that present that object, as shown in Figure 7.2. (The double-headed arrow indicates a one-to-many relationship.) Another requirement is that the displays should respond to a source change by updating only those portions that are affected by the change; i.e., an entire display unit should not have to be redrawn when one path value changes. The PathTable therefore identifies which components within the display unit are responsible for each path value. Finally, there should be no restriction on the number of times a path is represented within a display, so a one-to-many relationship exists between a path and the interested executors.

Since the ExecNotification table consists of two levels of mappings, the Interaction Manager uses two keys to find the appropriate Interaction executors when a database update occurs: 1) the updated source object and 2) the path within the source object whose value was updated. These pieces of information are provided by the Source-Update Manager as described in Section 7.3.

Aside from the SourceInterest Table, two other system tables are used to detect and report changes in source objects. The *ActivePaths* table keeps a record of the Outlines that were used to generate displays that are active; it also records a list of paths that are always of interest to each Outline. The *CurrentReports* table holds reports of updates to source objects. The reports are deposited in CurrentReports whenever the Source-Update Manager detects a change that can affect the displays. The ActivePaths and CurrentReports tables reside in the database and are managed by the Source-Update Manager.

7.1 Display Generator

The Display Generator's main responsibility is to create Layout and Interaction executors. As it does so, it adds information to the ExecNotification table. The following two subsections discuss implementation of the replication and construction phases for display generation. The final subsection discusses the generation of new formats during display execution.

Replication Phase

As stated in Chapter 6, the purpose for the replication phase is to initialize executors with information from an Outline spec and with other data specific to the source object supplied at Outline invocation. The latter data includes current path values and in some cases, the proxies representing those database values. In addition, any parameter values supplied at invocation are incorporated into the executors. Another main task in this phase is recording information in the ExecNotification table. In addition to the information described above, the Display Generator records data that helps to maintain the table during display execution, as described in the following paragraphs.

For each invocation of an Outline, the Display Generator enters a new PathTable into the ExecNotification table under the source object of the Outline invocation. Information for the path-to-executor mappings is collected as each executor is created through the GSI replication mechanism. When replicating an Outline spec, GemStone provides GSI with a linearized representation of the Outline that encodes the connections among the specs that making up the Outline. The replication mechanism uses this information to build a Smalltalk object (executor) having the same composition as the Outline.

For certain kinds of specs, several executors are generated per spec, resulting in additional paths of interest to a display unit, which must be recorded in a PathTable. In particular, an Iteration spec produces several Deferment executors, and paths are generated for each element of the collection being displayed. A DBRelConnect spec begets an extra path for each domain object in its Correspondence spec. For each path P leading to a domain object, the display unit is also interested in the path consisting of P plus an extra step: the relationship represented by the DBRelConnect.

To support maintenance of the ExecNotification table when format changes occur at runtime, the executors generated for a format option are tagged before being placed in a PathTable. The tag identifies which ChoiceMap contains that format option, and is used to find the appropriate executors for removal from the PathTable when the format option is replaced with another one. The paths of interest to a format option are likewise tagged since the Source-Update Manager must delete interest in those paths when the format option is changed.

Construction Phase

After the replication phase, the newly created executors hold all the information necessary to manage a display. Based on that information, the Display Generator extends the executors with various runtime objects needed to carry out the described functionality. Layout executors require capability to produce screen images; Interaction executors are extended with objects to interact with input devices, source objects, and other Interaction executors.

To complete the Layout executors, the Display Generator creates the Smalltalk views and other graphics objects that will draw on the display screen. For LayoutRect or LayoutLine executors, a Smalltalk Rectangle or Line object is created and made part of the executor's state. A Smalltalk ParagraphEditor is associated with a LayoutString executor to handle text display and formatting. The different kinds of ComposerLayout executors (Beside, Above, Around, and ViewOver) are also assigned a view in which they arrange the views or graphic objects for their subparts.

As the Display Generator builds the views and graphics objects, values in VisFeatureSet executors are consulted to set certain variables. In particular, spacing features are used to calculate sizes of the views for Beside, Above, and Around executors. Similarly, the border and color features in a VisFeatureSet determine those characteristics of a Smalltalk view. Thus, before the view and graphic objects are built, the Layout executor hierarchy is traversed to fill in any VisFeatureSet sub-objects that should be taken from parent Layout executors.

The runtime data needed to complete Interaction executors involves setting up connections among the Interaction executors to support event handling and their inter-communication. To support the operation of a Router executor, its subInteractions are examined to see what types of events each will respond to. Using that information, a mapping from event types to the subInteractions is created. Each Router is also assigned a BehaviorController that will receive the user inputs directed towards the screen area of the Router's `subject` Layout executor. The subInteractions of Coordinators receive a reference to the Coordinator so they can send it internal events. If the subInteraction of the Coordinator is a Router, the Router's subInteractions also receive a reference to the Coordinator.

To support the display of relationships through a Correspondence executor, the Display Generator gathers information on the domain and range objects involved with the Correspondence. The Display Generator determines which Layout executors represent a a domain and range object pair so that the objects' images can be visually related as defined in the Correspondence. This information is stored and used when reflecting changes in object connections.

The Display Generator creates a dictionary to record which Layout executor is responsible for rendering each domain object. The Display Generator uses information in a DBRelConnect executor to determine the range objects for the Correspondence's domain objects. The Layout executor for a range object is found by first consulting the ExecNotification table to obtain the Interaction executors for the range object. If the range object is being displayed in several places, the Display Generator looks for an Interaction executor that exists in the same display unit as the Correspondence executor. The subject of that Interaction executor is the Layout executor that renders range object's display image.

7.1.1 Executor Generation for New Formats

After a display has been created, a change in its format requires the generation of new executors. When the Display Generator creates executors for a new format choice, it looks for executors within the old format that should be reused in the new one. Reusing executors is beneficial not only for avoiding the cost of re-generation, but also for preserving any data or control state within those executors that should not be lost during a format change. The search for reusable executors is facilitated by a spec-to-executor mapping that includes all the specs used to generate a format; this mapping is updated whenever a new format is built. As in the replication phase for initial generation, the ExecNotification and SourceInterest tables are maintained to include the path interests of the newly created executors. When switching between formats, a display unit sometimes loses interest in certain paths and entries for those paths are deleted from both system tables.

A format that has been replaced with a new one is not stored for future use, due to two considerations. First, the alternate formats could take up excessive space in memory. Second, their executors would have to be kept up to date with changes in source objects, and this extra maintenance may be wasted if formats are not used again. Storing alternate formats could be warranted in specific cases where only a few options are needed and it is very likely that the alternate formats are used at later points in execution.

An executor is reused if it was generated from a spec that is shared by the old and new format specs. A ChoiceMap holds the spec-to-executor mapping, called the `specDictionary`. As each spec in the new format is being replicated, the `specDictionary` is checked, and if that spec is present, its corresponding executor is used within the new format. A dictionary lookup is already a part of GSI's replication mechanism: a dictionary that maps GemStone objects to Smalltalk objects is maintained so that multiple references to a GemStone object will not produce multiple copies within the generated Smalltalk object. To reuse existing executors, the Display Generator inserts the entries of the `specDictionary` into the replication dictionary before the new format's executors are created. After the new format has been replicated, the replication dictionary serves as the new `specDictionary`. Any dictionary entry holding a spec from the old format is removed if it is not a part of the new format.

Changes to the system tables are required if the old and new formats contain a different set of Path specs or defer to different Outlines. The ChoiceMap keeps a reference to the PathTable representing its display unit, so the PathTable can be updated to hold the revised set of Paths. Maintaining the ExecNotification table involves two parts. The first is adding information related to the new format choice, which includes inserting executors interested in newly added Paths and adding PathTables for display units within the new format. This information is added in the same way as during initial display generation. The second part is removing the information that pertains to the old format choice. The Paths whose entries should be deleted are those from the old format that were not reused in the new one. When the executor that is removed is a Deferment, the PathTable representing the Deferment's display unit must also be removed from the ExecNotification table. The Display Generator calls on the Interaction Manager to remove the PathTable, which requires the same procedure as used to delete information when a top-level display is closed.

The Interaction Manager informs the Source-Update Manager of the changes in the ExecNotification table, so they may be reflected in the SourceInterest table. The Display Generator sends a list of the new paths, and each is tagged to indicate which ChoiceMap it belongs to. The Display Generator also sends a list of identifiers for the PathTables that were removed from the ExecNotification table. The identifiers are used to find the InterestEntries that correspond to those PathTables, and they are removed from the SourceInterest Table.

After the executors for the new format have been created, the format's root executor must be placed into the position held by the old format's executor. Since it is possible that several other executors reference a format's root executor, the newly created executor "overlays" the existing one; i.e., the values of the new executor are copied into the instance variables of the existing executor.

As in the construction phase for the initial generation of displays, the new executors are assigned various runtime objects (e.g. Views and BehaviorControllers) used to manage the display. When inserting a new format, some of the existing runtime objects are reused. If the new format involves Layout executors, the VisFeatureSets and Views from the previous format are reused. The VisFeatureSet values are saved so that they can be used to fill in any unspecified feature values in the new executors. The existing View is used instead of creating a new one, since it already holds the proper position in the View hierarchy for the display's image. The View erases the image of its old contents before redisplaying the images for the new Layout executor.

In certain cases, additional steps are taken in the construction phase. When a format change shifts display focus, the source path of a DBConnect or DBRelConnect executor will be different from that of the previous format. Thus the new path value is retrieved and the DBConnect or DBRelConnect is activated, refreshing the display to reflect the new value.

7.2 Interaction Manager

The Interaction Manager handles all functions that involve graphical input and output. In addition, it processes events from various sources (user input, database, or internal) according to the event mappings taken from the Interaction specs. Thus, the Interaction Manager is responsible for the execution of semantic feedback, as well as communication with the database and the application (through the Control Manager). Those activities were generally described in Section 6.2.2. The following subsections focus on the implementation of specific tasks performed by the Interaction Manager: input handling; database-event handling, which involves maintenance of the system tables; and operations that refresh the display image.

7.2.1 Input-Event Management

The Interaction Manager's processing of input events is based on the event-management system already present in the Smalltalk environment. In Smalltalk, a display (or subdisplay) functions as a *Model-View-Controller* (MVC) triad. A Model is very similar to a source object; it is an object that is being represented by and manipulated through a display. A View is an object responsible for updating the display's screen image to represent the model. A Controller manages the keyboard and mouse input directed at the display. Specialized Controllers manage certain areas of user interaction such as menu operation or scrolling.

A BehaviorController is a specific type of Controller responsible for obtaining user input from the keyboard and mouse. A BehaviorController is created for each Router executor, and it produces Event objects that activate the Router, which in turn will activate other Interaction executors.

In the windowing system that manages MVC displays, a ControlManager object and a group of existing Controllers determine how user input is interpreted and how it affects the Views and Models. The ControlManager keeps a list of *scheduled Controllers* that correspond to the windows currently open in the Smalltalk environment. Only one scheduled Controller is active at a given time, and is called the *activeController*. Two messages in the Controller protocol are used in the selection of the activeController: "isControlWanted" defines the conditions under which a Controller wishes to gain control, and "isControlActive" defines the conditions that permit an activeController to remain in control. To select the activeController, the ControlManager polls its list of scheduled Controllers, searching for the next one that wants control, i.e., the first Controller for which "isControlWanted" returns true.

The activeController repeatedly executes its "controlActivity" method as long as it retains control. This method differs for different kinds of Controllers. The "controlActivity" for the Controller class polls the Controller's subordinate Controllers and passes control to the first one that wants control. A Controller's subordinates are the Controllers associated with the subviews of the Controller's View. (A subordinate Controller is never one of the scheduled Controllers.)

As one specific kind of Controller, BehaviorControllers participate in input handling as described above. A BehaviorController wants and keeps control as long as the cursor is contained within its View. BehaviorController's "controlActivity" checks input devices and generates Event objects that represent basic input events such as key strokes, button presses and clicks, and cursor movements (when the cursor enters or leaves the View of a particular Layout executor). The BehaviorControllers collectively manage the queue of the Events that activate the Interaction executors.

Thus, a single Controller subclass is used to implement different interactor behaviors; each BehaviorController is parameterized by the event mapping in an Interaction executor, which determines the behavior for responding to sequences of user input. This approach differs from standard technique for implementing MVC displays, where response behavior is represented as methods that are specialized to each subclass; thus a different Controller subclass is created for each desired behavior. Generating new classes for each type of response behavior is not suitable for executing Outlines due to the space cost and the time cost for compiling classes.

The ODDS prototype does not support window management capabilities such as moving, resizing, or collapsing displays. Such capabilities are often provided by a host environment on a given platform. An ODDS-generated display would be placed within a window provided by the host system.

Display Responses to Database Events

When the Interaction Manager obtains a report of database changes from the Source-Update Manager, it creates a database event for each affected path within the updated source objects. The new value for the affected Path is included in the event. The Interaction Manager consults the ExecNotification table to determine which Interaction executors should be activated with the database event; the executors are found by using the updated source object and affected

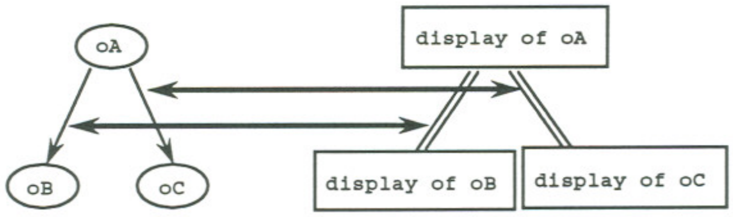


Figure 7.3: Correlation between Object Composition and Display Relations

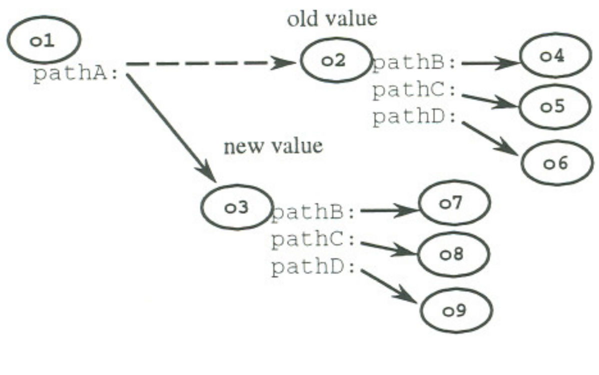
path as keys into the table.

When responding to source-object changes, the Interaction Manager must maintain the correlation between object connections and display-to-display relations; this correlation is represented by the thick arrows in Figure 7.3. This maintenance takes place when a path of interest to either a Deferment or a DBRelConnect receives a new value. For a Deferment, the display-to-display relation involved is display nesting. Recall that the source object of a (sub)display unit can change during execution because a Deferment is tied to a particular path rather than to the path's value. Thus, a subdisplay's source object changes when the path named in a Deferment is updated.

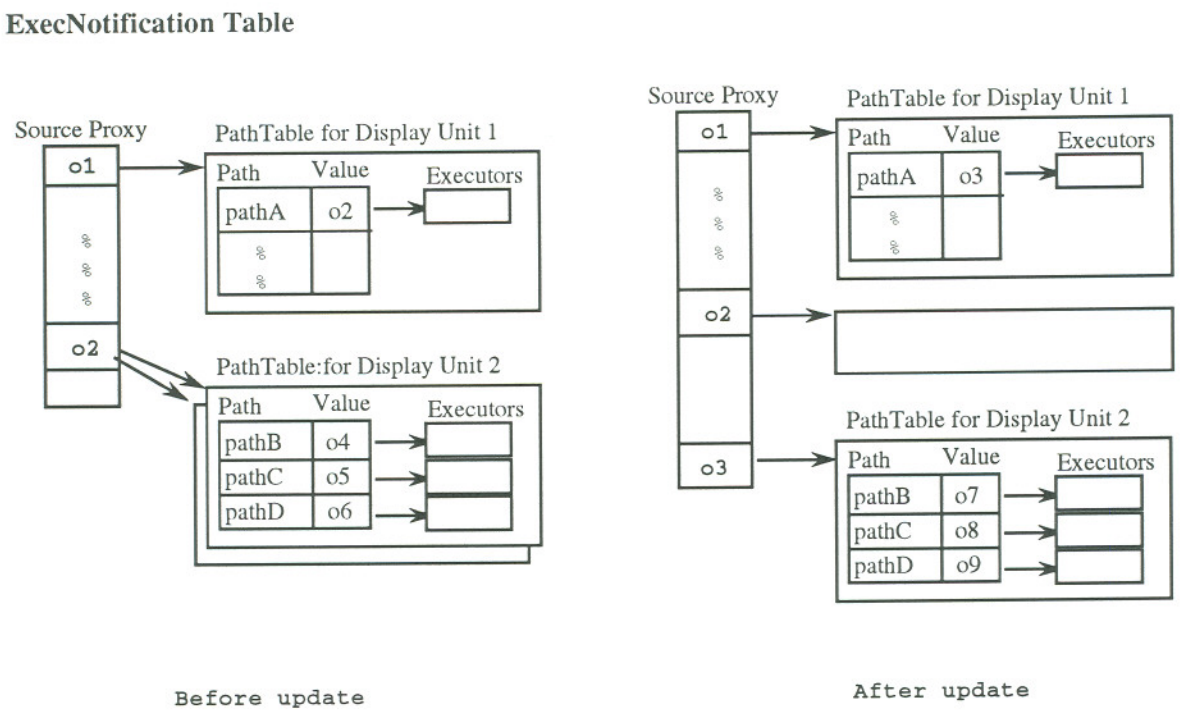
For example, suppose a display is generated for the source object `o1` in Figure 7.4a, where the invoked Outline contains a Deferment that describes a subdisplay for the path #(pathA). When the value for #(pathA) changes from `o2` to `o3`, the subdisplay's source object must also change from `o2` to `o3`. Consequently, any paths presented within the subdisplay also take on new values. For example, the value for #(pathA pathC) initially was `o5`, and its value changed to `o8` due to the change in the #(pathA) value. To deal with this situation, all the executors in the subdisplay's PathTable must be activated, because the executors now have different source objects. If any executors in the PathTable are Deferments, the process is repeated for the display units associated with those Deferments.

The system tables must be updated to reflect the change in a display unit's source object. In the ExecNotification table, the display unit's PathTable is removed from the the old source object's entry and placed into the entry for the new one (Figure 7.4b). The change is also reported to the Source-Update Manager, which adjusts the SourceInterest Table in a similar fashion.

For a DBRelConnect, the display relation that reflects object connections is the graphical line connecting the two displays.



(a)



(b)

Figure 7.4: Maintenance due to Composition Change

The Interaction Manager is also responsible for closing displays and removing the associated PathTables from the ExecNotification Table. Starting with the PathTable for the top-level display, the Interaction Manager recursively searches for removes the PathTables for nested display units.

Display Refresh

To carry out display updates, an Interaction executor changes values in its `subject` Layout executor, then sends it a message to redraw its screen image. The Layout executor updates its View based on the new values provided, then refreshes the View contents.

The Layout executors use various kinds of Views to manage their display images.

- ComposerLayouts use View objects, utilizing the support available in View methods for handling subviews. Displaying a View involves drawing its border and background, then sending the `display` message to all its subviews. The subview of a ComposerLayout's View are the Views created for the ComposerLayout's subparts. Not all Layout executors require that a View be created for them. In particular, the display images for LayoutRects and LayoutLines are produced through objects representing graphics primitives. A ComposerLayout executor includes a `graphics` instance variable, to hold graphics objects that were created for any of its subparts. Thus, rendering the ComposerLayout's image includes both drawing graphics objects and displaying a View and its subviews.

- LayoutText executors use Smalltalk ParagraphEditors, which provide the ability to compose a text string into several lines of text that will fit within a certain width. A ParagraphEditor has methods for updating the text string and for changing the width and number of lines of the display image. In addition, it can change the text's font, emphasis, and colors in response to changes in the LayoutText's features.

Whenever a Layout executor is instructed to redisplay its image, the values in its VisFeatureSet must be examined and used in refreshing the image. Recall that the described features include spacing, background and foreground colors, fill patterns for backgrounds and rectangles, bordering features, and text features such as font and emphasis.

A MotionOp executor performs display updates that translate the image of its `subject` Layout within the bounded area associated with its `boundLayout` value. The translation vector is determined by the guide coordinate and the `refPt` value. The guide point must be converted into a point within the coordinate system for the `boundLayout`'s View. The `refPt` value is used to derive a point also within that coordinate system, and determines whether the translation is restricted to the x or y dimension. The graphic object or view representing the MotionOp's `subject` Layout is translated using methods already provided in the View's class.

7.3 Source-Update Manager

The overall function of the Source-Update Manager is to detect and report database changes that affect the displays. Ideally, this would be accomplished without requiring the database programmer to insert notification messages into database methods. One approach for relieving the database programmer of this task is to detect object changes at the data-storage level. Existing techniques for maintaining an index over a data collection could be applied to this problem. These techniques detect when a particular instance variable in an object has changed so that the object can be repositioned within the index. For display notification, the DBMS could log the changes when detected, then report them to the display system.

However, this approach does not completely detect changes in an object's abstract state. Since some path values can depend on one or more stored values, these dependencies must be captured somehow to detect when the dependent path value has been updated. Hints from the programmer about dependencies among path values would be an immediate solution. Since it was not possible to modify GemStone's functionality at the storage level, this approach was not investigated further. Rather, the Source-Update Manager filters through the change notifications that are relevant to the displays, as described in this section.

7.3.1 The SourceInterest Table

As stated in the chapter introduction, the SourceInterest Table must maintain some invariants similar to those for the ExecNotification Table: the set of displayed objects and the paths displayed for each object must be kept current as the displays are updated. In addition, it must hold information for reporting changes to intermediate objects in multi-step paths.

The SourceInterest table maps a database object to a set of *InterestEntry* objects representing the display units that may be affected when the source object is updated. (See Figure 7.5.) An InterestEntry is similar to a PathTable in that it holds path information for the display unit it represents. An InterestEntry includes:

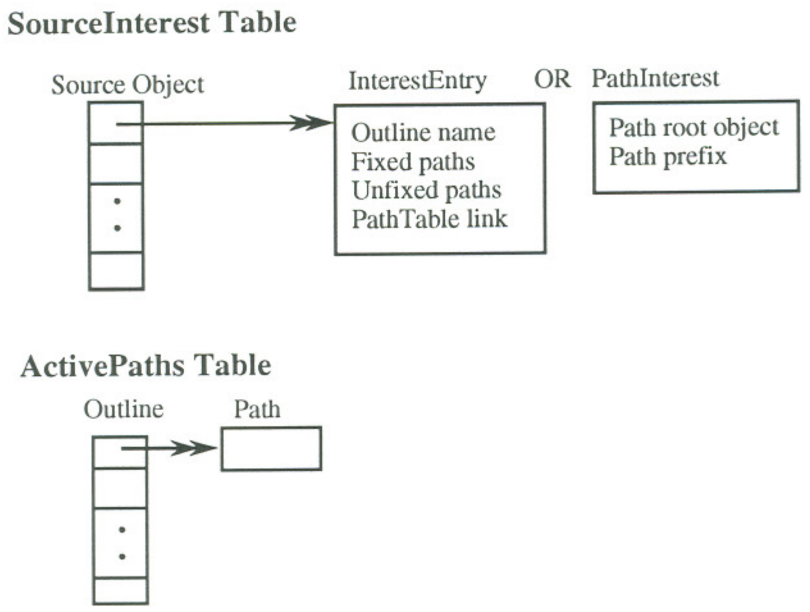


Figure 7.5: Structure of SourceInterest and ActivePath Tables

- The Outline from which the display unit was generated

- A set of *variable* paths that are of interest at some time during display execution. Variable paths are those found within the scope of a format choice. Since a given format may be replaced during display execution, the paths within its scope are not necessarily of interest to the display unit. The InterestEntry must record which paths are variable to aid maintenance of the Source-Interest Table. Whenever a ChoiceMap executor switches from one format to another, the paths of the old format are removed from the set, and those of the new one are added.

- A set of *fixed* paths that are not within the scope of any ChoiceMap and thus are always of interest to the Outline

- An identifier for the PathTable representing the same display unit as the InterestEntry. This association between the PathTable and InterestEntry is used for removing the appropriate InterestEntries when display units become inactive (i.e., when a display is closed or when a display unit is replaced due to a format change). The Interaction Manager informs the Source-Update Manager of which PathTables have been removed, and the Source-Update Manager removes the corresponding InterestEntry's.

The set of fixed paths for Outlines that have been invoked are contained in the ActivePaths table so that several InterestEntries may share this information, since a particular Outline may be used to create several displays.

Interest in source objects must also be recorded when a source object is at an intermediate position within a multi-step path. The SourceInterest Table contains *PathInterest* objects to record that an object is an intermediate one in some path of interest to a display. A PathInterest for an intermediate object holds two pieces of information:

1. The starting value (or root) of the path
2. A prefix of the path, up to and including the step within the intermediate object. This information is used to fill in a report of the source-object change, as described below.

7.3.2 Functions of the Source-Update Manager

The Source-Update Manager performs three main functions: adding entries into the SourceInterest Table for a newly generated display, creating reports of the changes in source objects, and maintaining the stated invariants for SourceInterest Table. Each function is described below.

First, the Source-Update Manager interacts with the Display Generator to create a display's initial entries in the SourceInterest and ActivePaths Tables. For each Outline invocation, the Display Generator provides the source object of the invocation, which will be a key in the SourceInterest Table, along with the other information to fill the InterestEntry. The Source-Update Manager collects the end-values for all the Paths and returns this information to the Display Generator. For any multi-step paths, the Source-Update Manager inserts PathInterests into the SourceInterest Table as described above.

Second, reports of source changes are created and provided to the Interaction Manager and the application. Recall that the database programmer has the responsibility of inserting notifications when an object is updated. A database method that updates the state of an object must include a message to the Source-Update Manager, specifying the changed object and the list of paths that were affected by the method. When the Source-Update Manager receives a notification, it checks the SourceInterest Table to see if the changed object is one that is being monitored, and if so, it checks whether the affected paths are of interest to any displays. A *ReportEntry* object is created for each changed path in an updated object, and holds the changed path and the new path value. The ReportEntry is entered into the CurrentReports Table, which is keyed by source object.

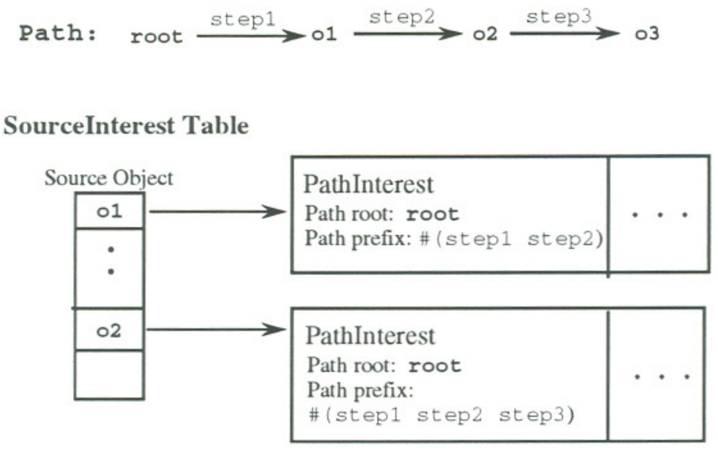


Figure 7.6: Path Interests for a Multi-Step Path

If an update occurs at an intermediate object in a path, it is reported as a change for the path's root object. Recall that a PathInterest is entered into the SourceInterest Table for each intermediate object in a path. If the affected path (i.e., one that was provided in a notification message) matches the last step in the PathInterest's prefix path, this indicates that the object chain is being altered. A ReportEntry is created, recording the root object as the updated object, and the PathInterest's prefix path as the changed path. Figure 7.6 provides an example, showing the PathInterests entered for objects along the path #(step1 step2 step3). Suppose `o2`'s value for #(step2) changes, a report will be created listing root as the updated object and #(step1 step2) as the affected path. Note that `o2` may have other InterestEntries in the SourceInterest Table; if a path named in one of those InterestEntries is updated, another report would be created, listing `o2` as the updated object.

Third, the Source-Update Manager maintains the invariants for the SourceInterest Table. Several circumstances can change the collection of objects that are of interest to display. For instance, when a top-level display is closed, its InterestEntry and the InterestEntry's for its subcomponent display units are removed from the table.

Another circumstance that affects the SourceInterest Table is the situation described in Section 7.3, where a path value is updated and is also the source object for a display unit. As mentioned, the Interaction Manager reports to the Source-Update Manager, providing the old and new values for the path and an identifier of the PathTable that changed position in the ExecNotification table. The corresponding InterestEntry under the old source object is transferred to the new source object's set of interests.

7.4 Control Manager and Application Communication Layer

The Control Manager's main role is to relay requests between the application and the display system. Together with the Application Communication Layer, the Control Manager handles the exchange of control, hiding the details of client-server communication from the other components. Thus the rest of the runtime system is not concerned with whether the application is a local process or on a remote machine, or whether it is a C or Smalltalk application. Other responsibilities of the Control Manager are to perform the initial setup of the system and to help coordinate the delivery of source-update reports to the Interaction Manager or application.

To initialize ODDS-Runtime so that it can accept connections from applications, the Control Manager sets up the process-coordination mechanisms as necessary and creates the central control thread. The central control thread becomes active when an application establishes a connection with ODDS-Runtime; the thread continues to exist as long as at least one application has a connection to the display system. If the central control thread terminates because all applications have closed their connections, a new central control thread is started whenever another application asks to open a new connection. (When ODDS-Runtime is initialized, a signal handler is set up to wait for new connections from applications.) Each new central control thread works from the same Control Manager object as the previous one, thus it is essentially a continuation of that thread. Similarly, the state of a display thread is held within the Interaction Manager, the executors, and the list of scheduled Controllers. The display thread is terminated whenever it submits a request to the Control Manager, and a new one is started whenever the display system activities are resumed. The new thread can continue display management as though it were the previous display thread.

Two mechanisms are used for coordinating between the system's threads and processes. First, the display thread and central control thread coordinate their activation through a Semaphore object [LaLonde91], which provides synchronization between Smalltalk threads. The basic concept is that a thread suspends itself by sending a `wait` message to a Semaphore; the thread is reactivated when some other thread sends a `signal` message to that Semaphore. The Interaction Manager communicates its requests by writing to a data location that is also accessed by the Control Manager.

Second, the ODDS prototype uses a socket, an interprocess communication mechanism available from the underlying UNIX operating system. Since the application may run as a separate process, Semaphores are not sufficient for synchronizing the application process and the central control thread. Service requests from the application are communicated by sending data through the socket. Both the central control thread and application process can suspend by waiting to read data from a socket.

The Control Manager also plays a role in reporting changes to database objects; it must inform the Source-Update Manager of which party initiated a database message, since the two parties have different requirements for the report they receive. Applications should receive reports on database changes made from user input, but not on the changes that it initiated. On the other hand, the displays should be notified of database changes resulting either from its own message requests, or from the application's use of the database. The display system may send database messages multiple times before the application regains control, so the changes from the messages must be accumulated in a separate report log for the application's use.

Chapter Summary

This chapter discussed the main system functions of ODDS-Runtime and how those functions interact. Nearly all the components are involved in maintaining the system tables that hold information needed to support complex display responses and dynamic representation. The required maintenance is centered on keeping a current record of the source objects relevant to the displays, the correlation between source objects and display components, and the correlation between object connections and display-component relationships.

Various system components are responsible for updating the system tables when necessary. The Display Generator performs maintenance whenever new executors are generated; the Source-Update Manager updates source-interest information when database messages are executed; and the Interaction Manager updates information on display relationships after format changes and also reports certain source-interest changes to the Source-Update Manager.

Chapter 8

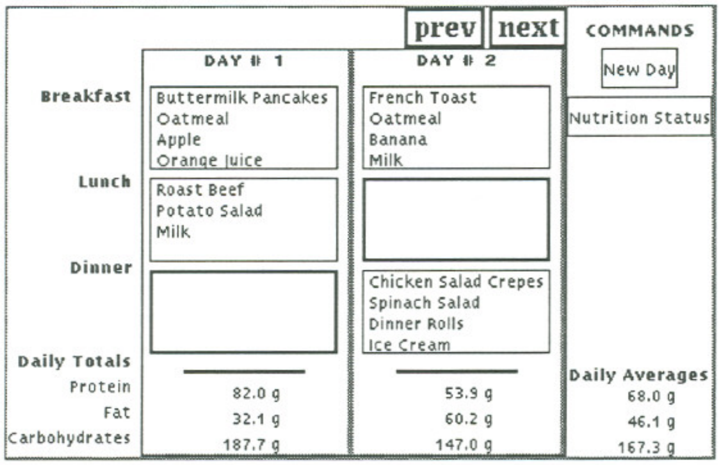
Experience and Evaluation

The displays introduced in Chapter 2 exhibit the kinds of complex display responses that ODDS is designed to support. Section 8.1 describes the Outlines that define those displays and points out some display features that were not easily specified, indicating places where improvements are needed in the specification framework. Section 8.2 discusses evaluations of ODDS regarding the expressiveness of the specification framework, the ease of creating and modifying Outlines, and the performance of ODDS-Runtime.

8.1 Outlines for Sample Displays

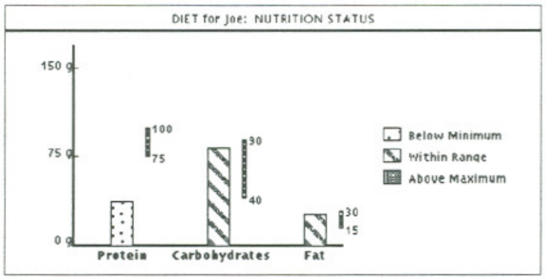
To help evaluate the ODDS prototype, several displays for the sample database in Section 1.4 were described and generated. The three top-level displays introduced in Section 2.2 provide different perspectives of a Diet object; they are shown in Figure 8.1. The schedule display (a) presents the DayPlans in a Diet's schedule, including the Meals and nutritional information for each DayPlan. Information on the Diet's nutritional content is presented graphically in a display for nutrition status (b), which is created by clicking on the button labelled 'Nutrition Status' on the schedule display. The meal-assignment display (c) provides the ability to update the connections of the DayPlans. Updating these source connections causes changes in the Diet's schedule and nutrient information, and those changes are reflected in the other Diet displays. In the schedule display, the newly-assigned Meal appears in the schedule table, and the daily totals and averages are updated. In the display for nutrition status, the bars change height to reflect the Diet's current nutrient amounts, and the bars' fill patterns change if necessary.

Another sample display, introduced in Chapter 5, presents the contents of a Recipe object and allows the user to choose between two perspectives, as shown in Figure 8.2. The one on the left presents sub-recipes as a separate display; the one on the right merges the ingredients

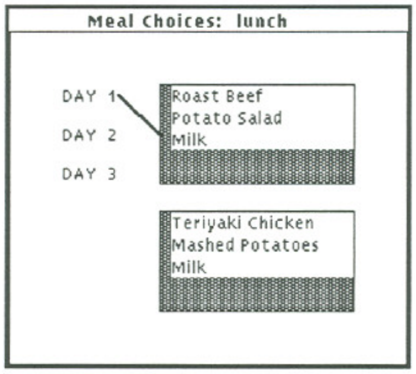


	DAY # 1	DAY # 2	COMMANDS
			New Day
			Nutrition Status
Breakfast	Buttermilk Pancakes Oatmeal Apple Orange Juice	French Toast Oatmeal Banana Milk	
Lunch	Roast Beef Potato Salad Milk		
Dinner		Chicken Salad Crepes Spinach Salad Dinner Rolls Ice Cream	
Daily Totals			Daily Averages
Protein	82.0 g	53.9 g	68.0 g
Fat	32.1 g	60.2 g	46.1 g
Carbohydrates	187.7 g	147.0 g	167.3 g

(a)



(b)



Meal Choices: lunch

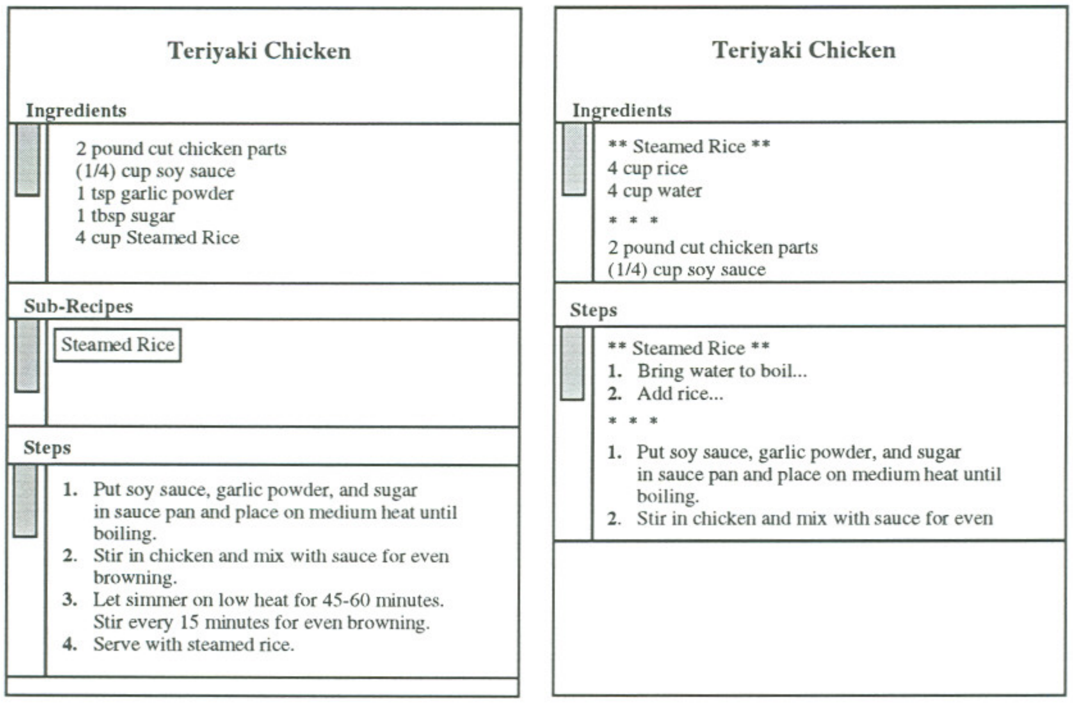
DAY 1
DAY 2
DAY 3

Roast Beef
Potato Salad
Milk

Teriyaki Chicken
Mashed Potatoes
Milk

(c)

Figure 8.1: Perspectives on a Diet Object



Teriyaki Chicken

Ingredients

2 pound cut chicken parts
(1/4) cup soy sauce
1 tsp garlic powder
1 tbsp sugar
4 cup Steamed Rice

Sub-Recipes

Steamed Rice

Steps

1. Put soy sauce, garlic powder, and sugar in sauce pan and place on medium heat until boiling.
2. Stir in chicken and mix with sauce for even browning.
3. Let simmer on low heat for 45-60 minutes. Stir every 15 minutes for even browning.
4. Serve with steamed rice.

Teriyaki Chicken

Ingredients

** Steamed Rice **
4 cup rice
4 cup water
* * *
2 pound cut chicken parts
(1/4) cup soy sauce

Steps

** Steamed Rice **
1. Bring water to boil...
2. Add rice...

* * *

1. Put soy sauce, garlic powder, and sugar in sauce pan and place on medium heat until boiling.
2. Stir in chicken and mix with sauce for even

Figure 8.2: Versions for Recipe Display

and instructions of sub-recipes with those of the main Recipe.

8.1.1 Display for Meal Assignments

The **MealChoices** Outline, which describes the meal-assignment display, has appeared throughout examples presented in earlier chapters. The display's functionality is explained in Section 4.2 and the specs defining that functionality appear throughout Sections 4.3 and 4.4. Figure 4.15 shows **MealChoices** in its entirety. In addition, a MatchMaker from **MealChoices** is used to illustrate MatchMaker execution in Section 6.2.2. Further details for defining the meal-assignment display are discussed here.

To specify the labels in the DayPlan displays, the **NumberedDay** Outline uses the special Path #(sequencePosition), which applies only to source objects within an ordered list; the value of this path is source object's position in the list. In **NumberedDay**, #(sequencePosition) provides the numbering for the DayPlans within the Diet's schedule. **NumberedDay** (Figure 8.3) and **MealDishes** define similar behaviors. Both include a response when the cursor is within the display image to indicate that subsequent input is directed towards that display. In **NumberedDay**, the response is to thicken the image's border, while in **MealDishes**, the

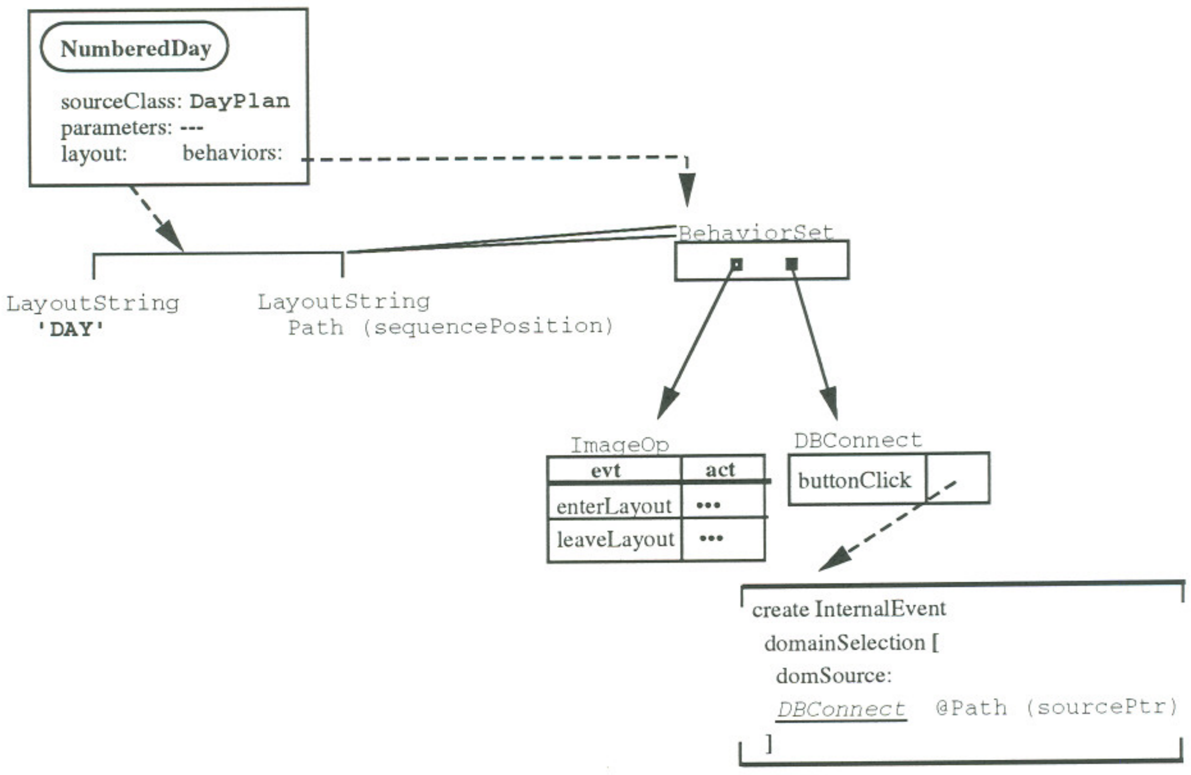


Figure 8.3: Outline for DayPlan Displays

response is to reverse the background and foreground colors.

One shortcoming in the display is that it does not respond to an addition or deletion of a DayPlan in the Diet's schedule. Presently, there is a restriction that the domain objects in a Correspondence remain fixed, because the implementation does not perform the necessary adjustments in system tables to handle changed domain objects.

8.1.2 Display of Nutrition Status

The **NutritionStatus** Outline describes the nutrition status display, deferring to the **NutrientData** Outline to describe each bar subdisplay. Each subdisplay presents the Diet's current daily average for either protein, fat, or carbohydrates. The current value is obtained from the NutritionLog at the #(currentNutrition) path within the Diet. The subdisplay also includes a marker that indicates the upper and lower bounds allowed for that nutrient, which are obtained from the NutritionLogs at #(dailyNeeds) and #(dailyLimits). The **NutritionStatus** display is not updatable through user input; it only reflects changes in source objects, which might be triggered by changes made either through the **MealChoices** display or by the application.

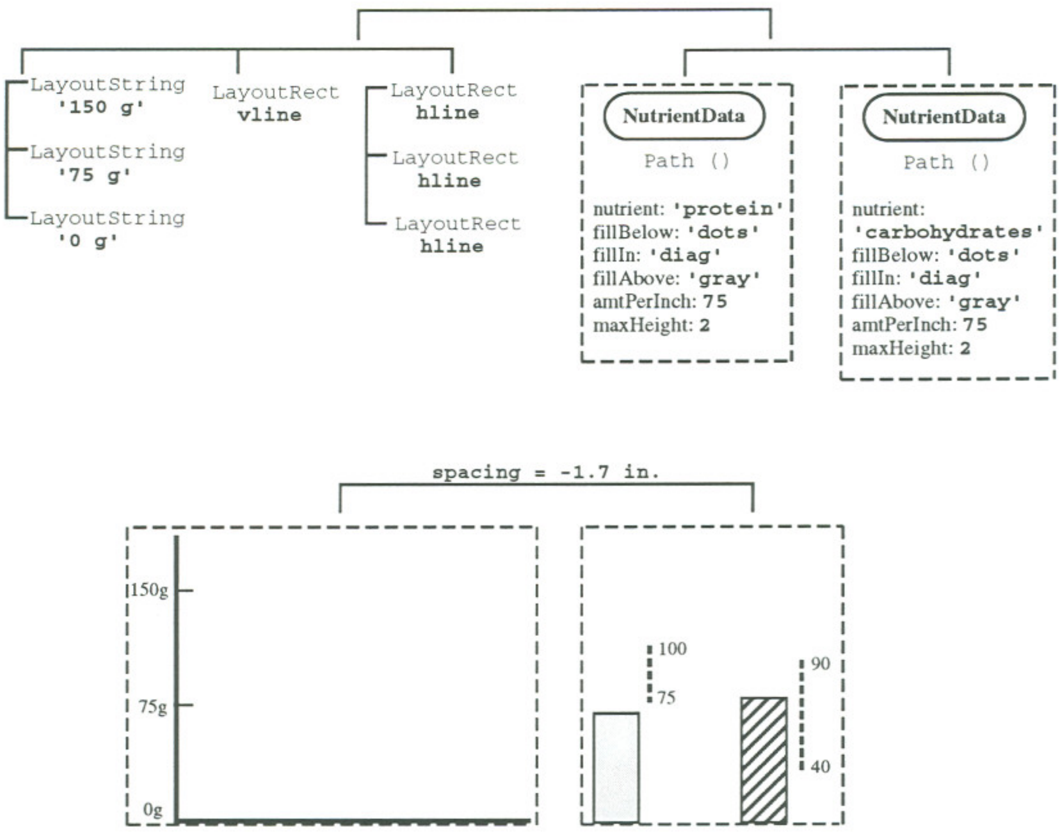
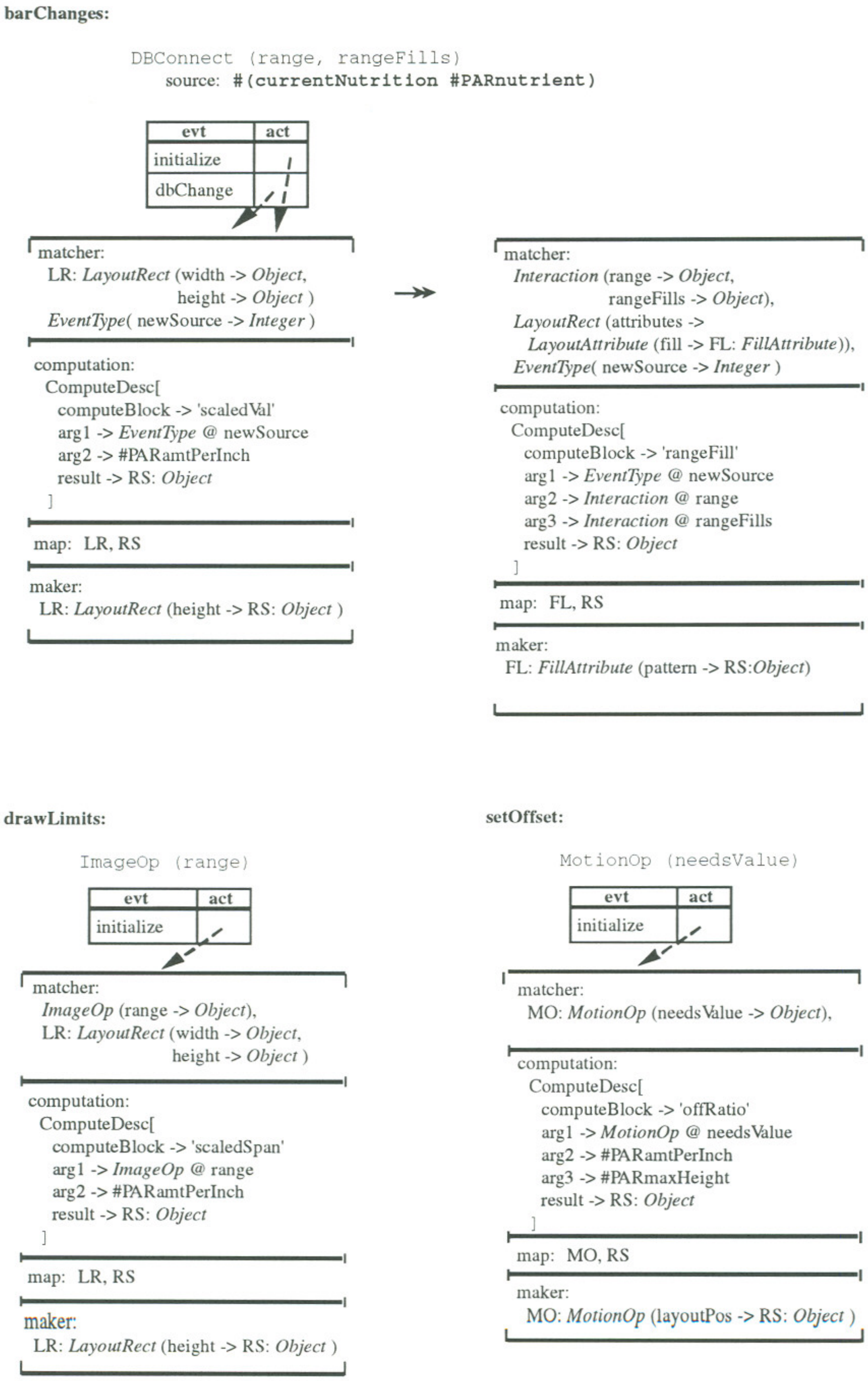


Figure 8.4: Layout Spec and Resulting Image for **NutrientData**

All the subdisplays have basically the same image and behaviors, but refer to a different path within the Diet source object. The **NutritionStatus** Outline supplies the fill patterns for each Deferment to **NutrientData**, matching the fill patterns displayed in the graph's legend. It also supplies a scale factor that matches the scale used for the vertical axis, which represents 75 grams with one inch; thus, 75 is supplied as the parameter value for *amtPerInch*. The maximum height provided to the **NutrientData** Deferments is the height (in inches) of the vertical axis. The supplied parameter values are shown in Figure 8.4.

Figure 8.4 also shows that the axis and the nutrient subdisplays are described as separate units that are composed with a Beside spec, and are made to overlap by defining a negative value for the the spacing attribute of the Beside spec. The Beside spec's alignment attribute is set to align its subparts along their bottom borders. (The default alignment is along the top borders.) The Beside spec that composes the subparts of the graph axes aligns them along the bottom as well.



barChanges:

```
DBConnect (range, rangeFills)
   source: #(currentNutrition #PARnutrient)
```

evt	act
initialize	
dbChange	

```
matcher:
  LR: LayoutRect (width -> Object,
                  height -> Object )
  EventType( newSource -> Integer )
computation:
  ComputeDesc[
    computeBlock -> 'scaledVal'
    arg1 -> EventType @ newSource
    arg2 -> #PARamtPerInch
    result -> RS: Object
  ]
map:  LR, RS
maker:
  LR: LayoutRect (height -> RS: Object )
```

$\twoheadrightarrow$

```
matcher:
  Interaction (range -> Object,
               rangeFills -> Object),
  LayoutRect (attributes ->
    LayoutAttribute (fill -> FL: FillAttribute)),
  EventType( newSource -> Integer )
computation:
  ComputeDesc[
    computeBlock -> 'rangeFill'
    arg1 -> EventType @ newSource
    arg2 -> Interaction @ range
    arg3 -> Interaction @ rangeFills
    result -> RS: Object
  ]
map:  FL, RS
maker:
  FL: FillAttribute (pattern -> RS:Object)
```

drawLimits:

```
ImageOp (range)
```

evt	act
initialize	

```
matcher:
  ImageOp (range -> Object),
  LR: LayoutRect (width -> Object,
                  height -> Object )
computation:
  ComputeDesc[
    computeBlock -> 'scaledSpan'
    arg1 -> ImageOp @ range
    arg2 -> #PARamtPerInch
    result -> RS: Object
  ]
map:  LR, RS
maker:
  LR: LayoutRect (height -> RS: Object )
```

setOffset:

```
MotionOp (needsValue)
```

evt	act
initialize	

```
matcher:
  MO: MotionOp (needsValue -> Object),
computation:
  ComputeDesc[
    computeBlock -> 'offRatio'
    arg1 -> MotionOp @ needsValue
    arg2 -> #PARamtPerInch
    arg3 -> #PARmaxHeight
    result -> RS: Object
  ]
map:  MO, RS
maker:
  MO: MotionOp (layoutPos -> RS: Object )
```

Figure 8.5: Interaction Specs for **NutrientData**

The major work in specifying behavior of the nutrition status display is within the **NutrientData** subdisplays. The behavior is described by three Interaction specs, labelled barChanges, drawLimits, and setOffset in Figure 8.5. The event mapping in the DBConnect barChanges holds two MatchMakers as the event response when the source value is updated. One MatchMaker describes the calculation of the bar's height based on the current amount of the nutrient; another determines the bar's fill pattern based on the relation of that amount to the limits set for the nutrient. In both cases, the new value for the nutrient's amount is obtained from a database event and is submitted as an argument of a computation block.

The ImageOp drawLimits and the MotionOp setOffset describe behaviors that are executed only at initialization, indicated by the fact that their event mappings contain only one entry, for the *initialize* event type. Initialization behaviors are needed because certain values within the Layout spec are calculated based on some source values. However, the specification framework does not provide a means to define computation within a Layout spec. The calculation of a bounds marker's height is described in drawLimits; the height is derived by taking the difference between the upper and lower limits for the nutrient, and scaling it using the scale factor. Calculation of the marker's position is described in setOffset. The Layout spec in **NutrientData** (not shown) defines the bounds marker initially to be aligned with the horizontal axis, and the operation specified in setOffset translates the marker vertically so that the bottom edge of the marker is aligned to a position that represents the nutrient's lower limit.

Some difficulty was encountered when using initialization behaviors to define the display's initial layout. In creating Outlines that have initializing Interaction specs, the designer must consider how the desired image may be produced by creating an initial image then performing some operation on that image. One factor that the designer must consider is the space requirements for the final image. In particular, if the initial image takes up less space than the desired image, then the View holding the initial image may not be large enough to accommodate the final image. To deal with this situation, the initial Layout spec can be made a subpart of a ViewOver spec that allocates enough space in which to draw the final image.

Another point to keep in mind is that the spatial relationships specified by ComposerLayout specs are not automatically preserved when the size of position of a subpart image is altered during display execution. For example, suppose the bounds marker appears with a mark (a short horizontal line segment) at both ends of a span bar. This image would be specified using an Above spec with LayoutRects as subparts. If the height of the span bar is initially set to zero and then updated after the correct height is calculated, the span bar would grow, but the top mark would remain in its original position. To achieve the desired result, the initialization operation must be specified as a format change, so that the executors for the Above spec will be regenerated and the top mark will be positioned correctly.

Another approach for positioning the marker is to specify some Layout spec below the bounds marker, then adjust the spacing attribute in the Above spec. This approach is not supported in the current implementation of ODDS because any operations on ComposerLayout specs were considered to be format changes. Thus, the positions of ComposerLayout subparts are recalculated whenever a new format is generated, but not in response to operations that update attribute values (which would be described with MatchMakers).

8.1.3 Diet Schedule

The schedule display exhibits behavior that adjusts display format in response to changes in a source object's composition. When the user clicks on the button labelled 'Add Day', a new DayPlan object is appended to the DayPlan array (the value for the Diet's 'schedule' connection). The display then responds by adding a subdisplay to the row of DayPlan subdisplays. The behavior to add a DayPlan subdisplay is defined implicitly though an Iteration spec that has the `isDynamic` element set to 'true'.

Another noteworthy feature in the diet schedule display is the generation of another top-level display. A SpawnDesc spec that defines this action has semantics similar to a Deferment spec. The Outline spec used to generate the display is chosen at runtime, and the choice is dependent on the type of the source object for which the display is created. In the **DietSchedule** Outline, the SpawnDesc spec defines **NutritionStatus** as the `outlineName`. Since `ExtNutritionLog` has a specialized Outline named **NutritionStatus**, the spawned display will differ depending on the Diet source object; in particular, whether its #(currentNutrition) value is a NutritionLog or an ExtNutritionLog. As noted earlier (Section 4.2) choosing deferred Outlines dynamically reduces the need to modify a deferring Outine (in this case, **DietSchedule**) in the event that newly added subclasses require changes in display requirements. Thus, existing Outlines are more likely to be reusable after such schema additions.

The **NutritionStatus** Outline for `ExtNutritionLog` provides an example of an Outline that defers to some Outline associated with the superclass of its (the deferring Outline's) source class. In creating the Outline for `ExtNutritionLog`, it was necessary to use a different Outline

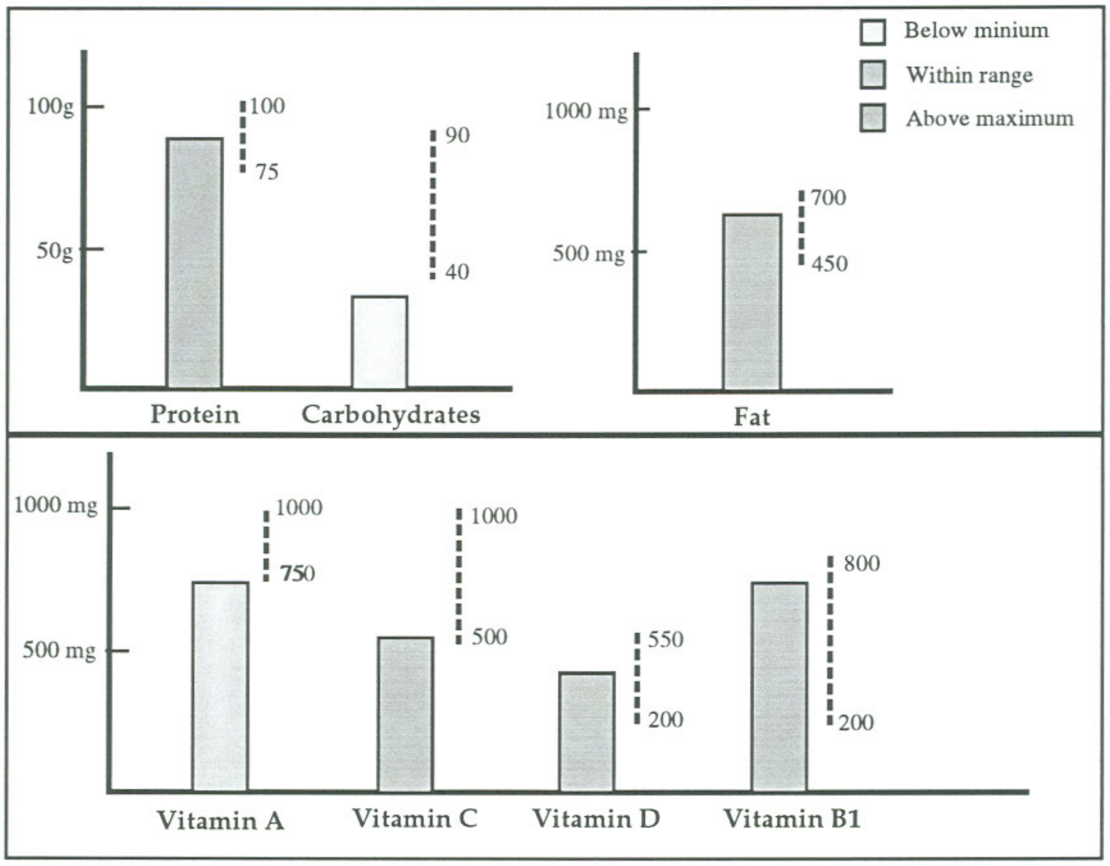


Figure 8.6: Display of ExtNutritionLog

name when deferring to the **NutritionStatus** Outline for `NutritionLog`, because the dynamic binding process would choose the Outline for `ExtNutritionLog` instead, resulting in a recursive deferment (and an infinite loop). To avoid this situation, a shallow copy[1] of the **Nutrition-Status** Outline for `NutritionLog` was created and given a new name, which was used as the `subOutline` instead of **NutritionStatus**. This experience points out that there are situations where a display designer will want to override the procedure for selecting the deferred Outline. One possiblity for addressing this need is to extend the specification framework with a specialized `Deferment` spec that would refer to a particular Outline spec directly, rather than holding an Outline name.

8.1.4 Recipe Display

The **RecipeData** Outline (Figure 8.7) defers to several Outlines that describe subdisplays for the ingredients, sub-recipes, and instructions in a Recipe. All the deferred Outlines have the class Recipe as their source class, indicated by the `subSource` path being the empty path in each Deferment. **RecipeData** illustrates two situations in which an Outline would defer to another Outline on the same source class.

The first situation is seen in the Deferments to the Outlines **IngredList** (Figure 8.8) and **RecipeSteps**, where the Recipe subpart being displayed is an array of items. The class `Array` is not an appropriate source class for these Outlines, because an array's elements are not restricted to any type, while **IngredList** and **RecipeSteps** are intended to describe displays of lists having a specific kind of element. Thus, **IngredList** and **RecipeSteps** each contain the path within a Recipe that leads to the array to be displayed. An alternative would be to create a subclass of Array, say `ArrayOfRecipeItem`, that can act as the source class for those Outlines.

The Deferments to **ScrollBar** and **ScrollView** exemplify the second situation, where the generated displays do not present any data from the source object. Rather, the activities of such displays are related to processing user input and changing the display image, and are independent of any data being displayed. However, since such a display is commonly used as a subcomponent of other displays, it is useful for its description to be in a distinct Outline to which other Outlines can defer. Outlines such as **ScrollBar** and **ScrollView** are defined with

[1] A shallow copy of an object `X` refers to the same objects that `X` refers to, rather than referring to copies of those objects. Thus, when making a shallow copy, only one new object is created.

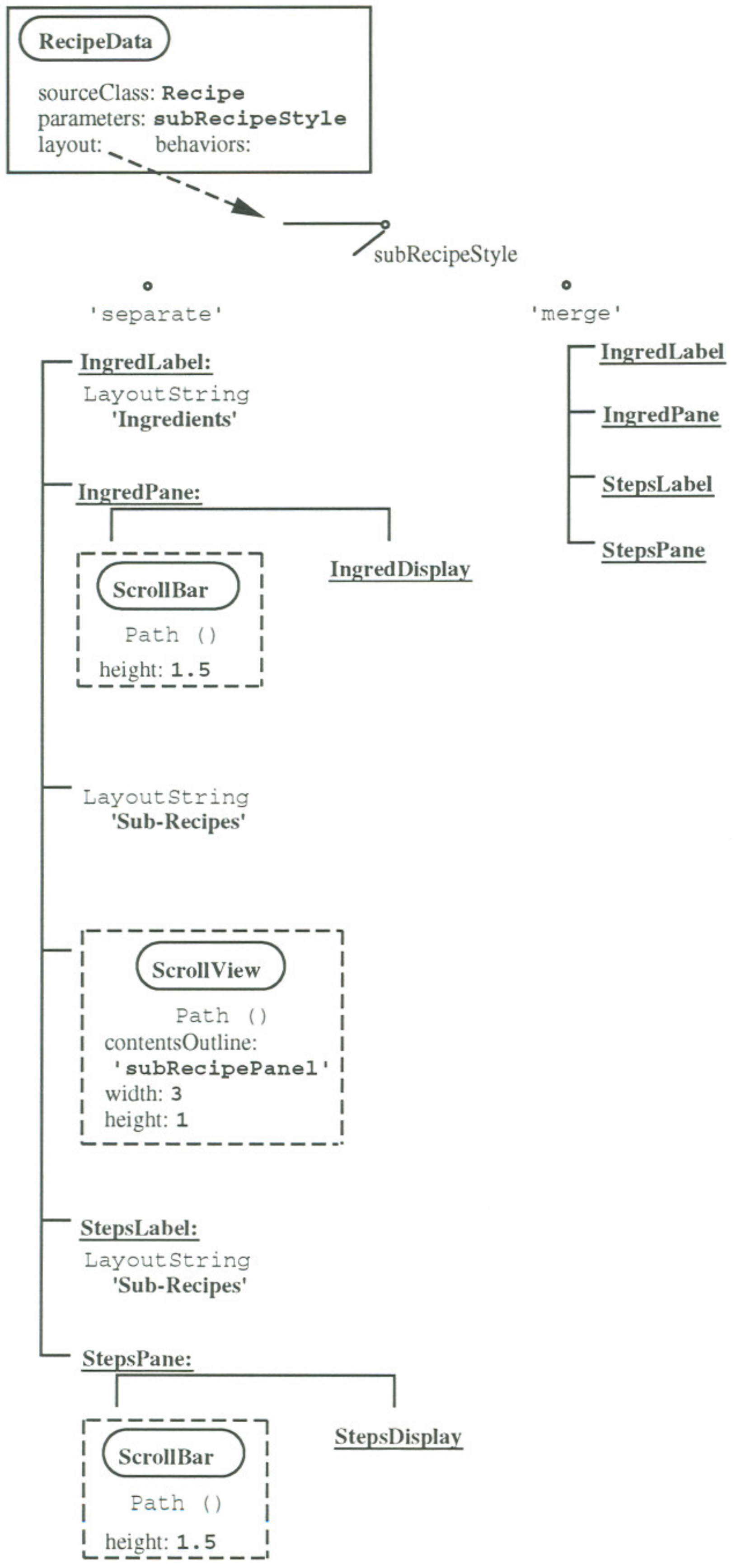


Figure 8.7: **RecipeData** Outline

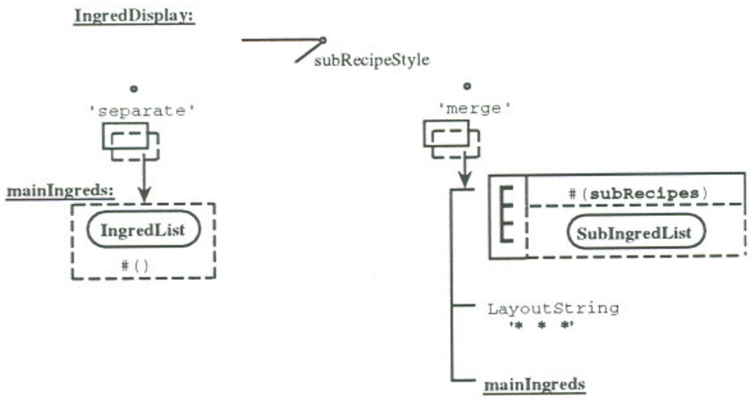


Figure 8.8: Ingredients Display

the Object class as their source class, thus they may be used to generate subcomponents within displays for any kind of object.

The **IngredList** Outline consists simply of an Iteration spec that defines **RecipeIngredient** (with source class `RecipeItem`) as the Outline for displaying each RecipeItem in the source array. The Iteration also specifies that the source array is obtained through the path #(ingredients). Although **IngredList** is very simple, it was made a distinct Outline so that another Outline, **SubIngredList** may defer to it. The **SubIngredList** Outline describes the display of the ingredients for sub-recipes in the merged version of the Recipe display. The sub-recipe ingredients are displayed in the same way as the main ingredients, except that the sub- Recipe's name is displayed as well, and thus **IngredList** is reused. A similar situation holds for displays of a Recipe's steps and the steps of its sub-Recipes, therefore the Outlines **RecipeSteps** and **SubRecipeSteps** are almost identical to **IngredList** and **SubIngredList**, respectively.

RecipeData also illustrates that ChoiceMaps may be nested. (The ChoiceMaps labelled IngredDisplay and StepsDisplay are within IngredPane and StepsPane, respectively.) The formats within the Steps and Ingredients panes are different for the two versions of the display, and the arrangement of the panes themselves is different as well. The ChoiceMaps that define these format changes are all dependent on the single `subRecipeStyle` parameter.

The pane presenting sub-recipes is described using **ScrollView**, which describes a scrollbar and viewport (the box in which an image is scrolled), along with the behavior necessary to coordinate the activities of the two components. The contents of the viewport, i.e., the image being scrolled, is described in another Outline whose name is submitted as a parameter value for **ScrollView**, as described in Section 5.1.2 (see Figure 5.2). For the sub-recipe pane, the contents are described by the **SubRecipePanel** Outline, which contains a ChoiceMap that depends on whether or not the subRecipes array is empty. Other parameters are the width and height of the viewport.

The ingredients and steps panes are also scrollable, but the specs for those panes (labelled IngredPane and StepsPane) do not defer to **ScrollView**. The framework of **ScrollView** is not compatible with the specs needed to specify format changes within the panes. **ScrollView** includes a ViewOver spec whose subpart is a Deferment representing a scrolled subdisplay. However, the definition of a pane's format change requires a ChoiceMap be placed as the parent of the ViewOver spec defining the scrolling viewport, as shown in Figure 8.8. The ChoiceMap could not be placed as a subpart of the ViewOver because the format choices for the pane contents are expressed with different types of specs (i.e., a Deferment spec for the separate version and an Above spec for the merged version[2]). Thus, instead of using **ScrollView**, the scrollbar's description is deferred to a **ScrollBar** Outline and **RecipeData** includes the viewport's description, as well as the specs for coordinating the scrollbar and viewport behaviors.

Difficulties were encountered when trying to create Outlines to describe other ideas for a recipe display. One idea for displaying sub-recipes was to use an icon in the ingredients pane to identify ingredients that are also recipes. Clicking on such an ingredient would cause its recipe to be displayed in the sub-recipe pane. To describe that behavior, a Coordinator would define communication from an Interaction spec in the ingredients pane to one in the sub-recipe pane, indicating which sub-recipe should be displayed. However, when sub-recipes are not being displayed separately (i.e., the 'merged' format is chosen), the sub-recipe pane is no longer present, and the executors for that behavior must be removed or somehow deactivated. The problem with implementing this version is that it requires coordination with an executor that exists only when a particular format is generated. A specification construct in needed to express that a certain behavior is valid only when a particular Layout format is chosen.

Another idea for the sub-recipe pane was to display each sub-Recipe in turn, using a 'NEXT' button to trigger the switch to the next sub-Recipe. This functionality is a format change in which the path leading to the displayed sub-Recipe is being changed; i.e., from #(subrecipes 1) to #(subRecipes 2) to #(subRecipes 3), etc. The specification of this format change requires

[2]Recall that format choices in a ChoiceMap represent different versions of the same spec, therefore the root object of all the choices must be the same type of spec.

a ChoiceMap over a Deferment spec, where each Deferment has a different Path spec as the `subSource` element. There is a problem with defining the set of possible format choices because the number of sub-recipes for a given Recipe object would be needed, but is not available since an Outline is defined at the class level. Thus, the format choices cannot be specified as a discrete set. ChoiceMaps should be improved to allow the set of choices to be described functionally rather than discretely.

8.2 Evaluations

8.2.1 Expressiveness of Display Functionalities

This section mainly addresses the specification framework's expressiveness regarding display responses to changes in source objects. We concentrate on this aspect of the framework because of its key role in achieving directness in displays. As stated in Section 3.1.1, semantic feedback can be viewed as the translation from a source object's abstract state to an external representation, occurring whenever the object's state changes. The types of translations supported in ODDS are:

1. mapping a path value to a visual representation
2. calculating and displaying a value from path values in the object, for example, calculating an average or sum
3. updating display attributes (such as color, fill pattern, or spacing) based on path values
4. mapping a connection between objects into a visual representation
5. coordinating multiple display changes to reflect a particular change in the object

For each type, I describe how it is supported by the specification constructs in ODDS, drawing from experience in creating the sample displays. In addition, areas that require future work are pointed out.

Regarding translations of Type 1, a path may represent either an attribute or a relationship. Mapping attribute values such as strings, numbers, or symbols into their textual representation is easily specified, using a LayoutString spec that contains a Path spec leading to the value. Changes to the path value are reflected in the display if a DBConnect spec is attached to the LayoutString. No entries need to be added to the DBConnect's event mapping because the semantics of a DBConnect includes a default behavior to perform the translation if its source object is an attribute value and is updated. A bitmap can also be considered an attribute value, however translations from bitmaps to displayable images are not currently supported in ODDS. The specification framework could be extended with a LayoutBitmap class to provide this capability.

If the path represents a relationship, the path value being displayed is a complex object. Mapping a complex source object to a visual representation is expressed using either a Deferment spec or a DBConnect whose `subject` is a ComposerLayout spec. If a Deferment spec is used, all components within the generated subdisplay are updated automatically to reflect the source object's state. If a DBConnect spec is used, its event mapping will explicitly describe the display changes that will reflect the path values within the complex source object. A typical action is to send out internal events to interactors or subdisplays that perform translations for each path value.

The current implementation of ODDS imposes a limitation on the framework's expressiveness. Currently, the event types that represent source-object changes do not indicate which path within the object has been changed. If the designer wants to describe different event responses based on which path(s) were affected, he or she must define behavior to determine that information. Information on affected paths is in fact supplied by the Source-Update Manager, but presently is available only to the Interaction Manager for use in activating the appropriate Interaction executors.

Translations of Types 2 and 3 involve computations using source object values, producing a value that is either to be displayed or is used to update a graphical attribute in the display. In the specification framework, execution of computations may be specified through ComputeDesc, MessageDesc, or Path specs. MatchMakers complete the specification of the calculation, since they specify where arguments come from and where the result is placed.

The specification framework allows for calculations to take place either within or independently of the database. ComputeDesc specs describe computations that are concerned only with the operation of the display, drawing little semantics from the source objects. Thus, ComputeDescs support the mapping from source values to graphical data, for example, the distances that represent the current amount and the upper and lower limits for a nutrient (in the **NutrientStatus** displays). ComputeDescs can also support computation of displayed values, but in most cases, derived values shown in displays are computed by database methods that have been defined as part of the database semantics. Examples of derived values defined by database messages include the nutrient information for a DayPlan and the list of sub-Recipes in a Recipe. MessageDescs and Paths represent the execution of database methods, thus supporting the presentation of derived values.

Type 4 translations update a display based on the state of object connections or on the occurrence of certain conditions regarding connections. Information derived from connections might be in the form of a simple value, e.g., the number of connections for a multi-valued relationship; this data would be presented using translations of Type 1 or Type 2. The connection itself might be translated to a spatial relationship such as display nesting or to some graphics such as a line connecting displays.

Since paths represent relationships as well as attributes, specifying translations that are triggered by composition changes is supported by DBConnect and DBRelConnect specs as described Type 1 translations (for attribute changes). Specifying that display actions are triggered by specific conditions in object state is more complicated, since the designer must decide when to check for the condition, and specify the check as part of the display's behavior. In the sample displays, examples of state conditions that affect the display are 1) whether a RecipeItem includes a form of preparation (e.g., chopped or diced), 2) whether a path in a DayPlan is set to a Meal or to nil, and 3) whether a nutrient's amount is within the desired range.

Translating an object connection to a visual representation can be expressed through a Correspondence spec or a dynamic Iteration that reflects changes in the number of elements of its source object (an array). Such a translation also can be supported by ChoiceMap specs that define alternatives for Layout or Deferment specs, if the designer wishes to change the subdisplay arrangement in response to changes in object connections.

As noted when discussing the Recipe display (Section 8.1.4), there are situations where the expressiveness of the ChoiceMap spec is not sufficient. In particular, the format choices are presently defined as discrete entities, but it is sometimes useful to define the set of format alternatives as a function. The example given was a subdisplay that can shift focus between elements of a source array. If the format alternatives are described discretely, the only difference between them is the source path; i.e., the spec for each alternative defines its source path to be a different position in the array. In this example, the set of alternatives could be described more compactly as a generalized spec that treats the source path as an argument.

Translations of Type 5 involve execution of several coordinated display updates triggered by a single source update. Several mechanisms are available in the the specification framework to achieve coordination. One such mechanism is to define several actions as a sequence of responses to a single event. For example, in the display of nutrition status, two features in the display can change when the nutrient amount changes: the bar's height and its fill pattern. Another coordination mechanism is to define the creation and forwarding of internal events among the subInteractions of a Coordinator executor.

Another way to coordinate display responses is to rely on the update detection mechanism to report additional source updates caused by a certain triggering update. For example, in the diet schedule display, changing one of the Meals in a DayPlan can affect all three nutrient amounts for the DayPlan, and consequently the daily averages (of nutrients) for the Diet can change as well. To define these coordinated display changes, the display designer would specify the appropriate display response for the individual source updates, and at runtime all the related display changes will be executed.

However, the specification framework lacks the expressiveness to define the execution order for multiple display changes that are triggered by a single message send. In the example just given, one would expect to see the display changes occur in a certain order: first the changed Meal would appear, followed by the updated nutrient values for the DayPlan, and then those for the whole Diet. Currently the order of display changes is dictated by the runtime system.

The specification framework has limited expressiveness with respect to input-event handling and image description since those aspects were not the main focus for the research. In particular, some limitations were encountered in specifying the test displays.

With respect to input handling, the framework supports a small set of input event types and a simple model for describing event dispatch, represented by Router specs. One limitation concerning Routers is they must be defined over non-intersecting areas of the display. It is not possible to associate a Router with a ComposerLayout spec if another Router is associated with one of the ComposerLayout's subparts. Thus, if the designer wishes to have an event type recognized over the entire image of the ComposerLayout, the event type must be defined in the event mappings of each of the ComposerLayout's subparts. To address this limitation, the semantics of a Router spec must include some convention for the order in which overlapping Routers receive an input event; e.g., in order of increasing size of the screen area associated with a Router. In addition, the Router spec should include an element to define whether an input event is allowed to propagate to subsequent Routers in the ordering. ODDS-Runtime would have to be extended to check for overlap in the screen areas of Router executors and record the ordering in which they receive events.

Regarding image description, it is not possible to specify that a Layout spec's image be aligned to more than one thing. An example is the value bar in the nutrition status display, whose bottom edge is aligned to the x-axis. The bar is also centered with the nutrient's label. However, one of these alignments needs to be "wired in"; i.e., the proper spacing has to be calculated by the display designer. To address this need, the specification framework could be extended with a LayoutGrid spec, a new kind of ComposerLayout spec for defining positioning and alignment in two dimensions.

8.2.2 Creating Outlines

This section describes experiences from the efforts to create Outlines. This type of evaluation helps to see how certain aspects of the framework affect productivity.

Outline Creation through Programming

In my usage of ODDS, Outlines were created by writing code that creates spec objects and enters the desired element values, thus building up the composition of Layout and Interaction specs as necessary. The code is written in OPAL, GemStone's language for data definition and manipulation. Using a programming language as the only means to create objects has several drawbacks. First, there is no convenient way to access specs within an Outline once the creation code has been executed. Thus, modifying features of the display description is typically accomplished by changing and re-executing the code, thus re-creating the entire Outline. This practice usually requires less effort than writing new code that navigates through the Outline to reach the particular object or objects that need to be changed. Yet it can cause a large amount of needless activity if the desired modification is small and the Outline is large. Second, although the activity described by the code consists of building up objects and is fairly straightforward, it is sometimes difficult to visualize the structure and content of the display description from reading the code, especially as the specs get larger. This difficulty increases the effort required to write the code initially and to modify it for later iterations of design.

These drawbacks point to the need for a graphical editing tool that displays and manipulates specs. A display of an Outline can give the designer access to points within the Layout and Interaction specs, enabling values to be updated without having to re-create the entire Outline. The ability to visualize the specs as they are being created can help the designer recognize similarities in different specs (or different parts of the same spec), which signal an opportunity for reuse. Because Outline specs are database objects, a graphical editing tool for specs could be developed using support from ODDS.

Support for Modularity and Specification Reuse

Typically, the display designer will have to go through several iterations of design before reaching the final version of a display specification. Modularity within Outlines reduces effort during iterative design by localizing the effects of making a change in an Outline. If a change to an Outline is known to affect a specific portion or aspect of the display, the designer can verify the effects of the change more easily. The ODDS construction model (Chapter 4) provides guidelines in understanding what functionalities are described in the specs, thus making it easier to modify them as needed.

One type of modularity supported in ODDS is the separation between the layout and behavioral descriptions of a display. The contents of Layout specs can be adjusted without changing the behavioral descriptions in the Interaction specs, and vice versa. For example, one might wish to add icons to menu items or to a title bar, but keep the behavior already defined for those components. Conversely, the behavioral description for a display might be changed to coordinate differently with other displays, while the visual description remains unchanged.

Another type of modularity is provided by using Deferments to specify subcomponents of the display. For the most part, the content of the deferred Outline can be changed without having to change the deferring Outline. However, in some cases the deferring Outline might be closely coupled with the deferred Outline when specifying coordination in their behavioral descriptions (through internal event types). Use of Deferments also simplifies modifying the Outline to be used in generating a subdisplay, since the designer only needs to change the `subOutline` element of a Deferment. In addition, Deferments reduce the effort of maintaining Outlines when the database schema changes such that instances of a different class might be referenced by an Outline's source object. Existing Outlines do not necessarily have to be modified, since the `subOutline` in a Deferment can represent different Outlines for different source classes.

The ability to reuse existing specs when creating new Outlines saves the designer the effort in developing a new spec to fill his or her needs. The development effort often includes corrections in the creation code's syntax and in the meaning of the specification, i.e., corrections to ensure that the spec produces the expected appearance and functionality in the display. For example, errors in display behavior may result from omitting the necessary event types in an Interaction spec's event mapping. Reusing an Outline that is already known to be correct or copying such an Outline and making small modifications to it requires much less effort than developing an Outline from scratch. Thus, reuse can be a significant factor in increasing productivity.

The effort required to reuse display descriptions includes defining components with the intent of including functionalities that are expected to be used in several displays, so that a component has good potential for reuse. A characteristic that makes a component reusable is that it is easily adapted to new contexts, thus reducing the effort to reuse the component.

The modularity supported in ODDS promotes creation of reusable components because a spec or spec fragment represents a logical unit that can easily be extracted and plugged into different contexts. Any existing Layout, Interaction, or Outline spec is a candidate for reuse. The ability to parameterize Outlines is another feature that enhances reusability of Outlines. A parameterized Outline is easily adapted to various contexts, since the display generated from it can vary without having to edit the Outline; the variation is specified by supplying different parameters values when the Outline is invoked.

The framework provides several mechanisms for integrating a reused component into another Outline, which were described in Section 5.1.2:

- Copying a spec fragment and editing it for a different context, e.g., changing the Paths within the spec to make it suitable for a different source class
- Outline composition through Deferments
- Using an Outline from a superclass, either as additional Outline for the subclass or as a deferred Outline within an Outline defined for the subclass

8.2.3 Implementation of Runtime System

This section discusses observations on certain aspects of the runtime system's implementation and performance. When designing the prototype implementation, emphasis was placed on obtaining the desired system functionality rather than on optimizing performance. Thus some modifications for improving system performance are suggested here, based on examining the present operation of the prototype system. Recall that the prototype consists of a Smalltalk application and a GemStone database in a configuration where the two communicated over a local-area network. The Smalltalk portion was implemented using Tektronix Smalltalk-80 version TB2.2.2a on a Tektronix 4317 workstation running the UTek 3.1 operating system. The database portion was implemented with version 2.0 of the GemStone OODBMS and executed on a Sun 4/75 with SunOS 4.1.3.

Outline	Generation Times (seconds)			% Time in Replication
NutritionStatus	46.4	51.8	57.7	91%
MealChoices	49.4	50.8	51.8	92%
RecipeData	55.4	56.6	60.2	94%
DietSchedule	84.2	95.2	97.5	91%

Table 8.1: Display-Generation Times

One aspect examined was the time required to generate the executors when the display is initially created. The generation time for the sample displays is about 1 or 2 minutes, as indicated in Table 8.1. The times shown are the total elapsed time for generation (minus the time to record the time measurements in a transcript), and thus include underlying activities such as garbage collection, memory swapping and network communication. Five measurements were taken per display, and the table shows the minimum, maximum, and median values. When taking these time measurements, a garbage collection was performed immediately before each display generation to minimize the time spent for garbage collection during the generation.

Since the generation times are much higher than desired, further analysis was performed to determine the time required for completing the subtasks within the generation process. As shown in Table 8.1, over 90% of the total generation time is spent in the replication phase. Recall from Section 6.2.1 that the replication phase merges data from Outlines and source objects. The Display Generator first replicates and collects information for the top-level Outline, then processes each of the deferred Outlines in the same way.

The values in Tables 8.1 and 8.2 show that the time for display generation tends to increase when a larger number of executor objects are created during the replication phase. An exception to this trend is seen in the **NutritionStatus** and **RecipeData** Outline, since fewer objects were generated for **RecipeData**, but generation took longer. Note that the number of generated objects for the two Outlines does not differ greatly (the number for **Nutrition-Status** is 7% higher), but number of deferred Outlines is much higher in **RecipeData** than in **NutritionStatus**. In the table, the total number of deferred Outlines refers to all the invocations of Outlines that occured when generating the top-level display, including multiple invocations of an Outline that is named in an Iteration. The data for the two Outlines indicates that processing many deferred Outlines in the replication phase can also have an impact on the

Outline	Outline Size (# GemStone objects)	Source Objects	Total # Deferred Outlines	Total # Replicated Objects
NutritionStatus	122	1 Diet 3 NutritionLog	3	932
MealChoices	205	1 Diet 2 DayPlan 2 Meal 8 Food	13	777
RecipeData	210	1 Recipe 5 RecipeItem 1 (sub) Recipe 4 String	16	871
DietSchedule	393	1 Diet 2 DayPlan 4 Meal 15 Food 3 NutritionLog	32	1353

Table 8.2: Sizes of Outines and Generated Displays

generation time.

To illustrate the connection between number of generated objects and generation time more clearly, Table 8.3 shows time measurements for certain subtasks involved in processing each Outline individually i.e., not counting the time to process deferred Outlines. The first subtask is obtaining a *traversal report* for replicating an Outline. The traversal report encodes the composition and values of an Outline in a linearized form. It consists of a collection of entries, where each entry represents a GemStone object in the Outline. An entry records a GemStone object's connection to another GemStone object by identifying the entry that represents the

		Subtasks (time in ms)		
Outline	Size	Create traversal report	Executor building	Get path values
FoodName	24	227	600	699
NumberedDay	84	836	2527	642
MealDishes	106	1042	2903	964
NutritionStatus	122	1638	5684	990
RecipeData	210	2674	9843	959
NutrientData	270	2812	9876	1383
DietSchedule	393	2970	16470	3604

Table 8.3: Breakdown for Replicating Individual Outlines

referenced GemStone object. Both subtasks involve requests to GemStone, thus the times shown include the time for network communication between Smalltalk and GemStone and for GemStone to perform the requested operation. The second subtask is the most time-consuming; it is to build the Smalltalk executors from a traversal report. The third subtask is retrieving a set of path values from the source object.

The data in Figure 8.3 indicates that the time spent for the first two subtasks increases as the Outline size increases. Producing a traversal report includes requesting data from GemStone and processing the returned data stream to form the traversal report, which entails visiting each packet in the stream. Since the Outline size determines the size of the data stream from GemStone, it directly affects the time to create the traversal report. As the replication mechanism builds the executor's composition, the traversal report is repeatedly searched to find some entry representing a referenced object. Since the size of the report affects the search time and the number of searches, it affects the total composition-building time.

Since the number of replicated objects is a major factor in generation time, the performance for generating displays can be improved by reducing that number. As one approach, the generation process could be modified so that an Outline is replicated only when it is first invoked, and the generated executors are cached by the runtime system for use in subsequent invocations of that Outline. This approach reduces generation time since creating a deep copy of a Smalltalk object is much faster than replicating a GemStone object. Table 8.4 shows the time required to make such copies for some of the Outlines. To create new executors for subsequent invocations, the Display Generator must also have to store information on each Outline's Paths, Deferments and Iterations, and any such information that, in the current implementation, is gathered as the Outline is being replicated. This information is needed to merge source values into the proper Layout and Interaction executors, and to create entries in the ExecNotification and SourceInterest tables.

Caching the executors for an Outline (and its deferred Outlines) optimizes display generation only if the Outline is invoked multiple times. Therefore, before caching an Outline's executors, the Display Generator could check for certain conditions that indicate multiple use of Outlines. An example of such a condition is when an Outline is named within an Iteration spec, and thus is expected to be used multiple times. Another such condition is when one Outline defers to another Outline several times. The sample displays provide two examples of this situation: the **NutritionStatus** Outline defers to **NutrientData** several times, and **MealsColumn** defers

Outline	Time for Deep Copy of Executors (ms)
FoodName	65
NutrtionStatus	129
MealChoices	351
DietSchedule	380

Table 8.4: Times for Deep Copy of Executors

to **MealDishes** for the breakfast, lunch and dinner of a DayPlan.

Another approach to reducing the number of replicated objects is to perform some optimizations within the GemStone database before Outline specs are sent to the Display Generator. One possible optimization is detecting identical sub-objects in the Outline, which can be merged into a single shared object. Another possibility is to detect multiple Deferments to the same Outline in the database, rather than having the Display Generator perform the detection, as suggested above. A list of Deferments invoking a common Outline could be sent to the Display Generator, along with the source values needed to generate the Layout and Interaction executors.

Performance for display generation can also be improved by changing the software and hardware platforms on which the runtime system is built. In version 2.5 of GemStone, the GemStone-Smalltalk Interface can be executed as part of a Gem process, thus eliminating the need to communicate with the Gem through sockets. This capability would reduce the time required to obtain traversal reports and path values during the replication phase. Version 2.5 also features a replication mechanism with improved performance. As seen in Table 8.5, the elapsed time to create executors from Outlines is several times shorter for Version 2.5.

Although the time spent in garbage collection and virtual- memory management could not be measured, observing the duration of Smalltalk's garbage collection cursor and the amount of disk activity during display generation indicates that these activities take up a significant amount of time. Thus, choosing a machine with larger main memory would provide some improvement on system performance. The workstation running the ODDS prototype had 8 megabytes of main memory.

Another aspect of system performance that could be improved is the time required to draw displays that have nested subdisplays. Although database changes often affect a single-unit

Outline	Time for Creating Executors (ms) (Subtasks 1 + 2)	
	Version 2.0	Version 2.5
FoodName	827	118
NumberedDay	3363	425
MealDishes	3945	379
NutritionStatus	7322	786
RecipeData	12517	1668
NutrientData	12688	2359
DietSchedule	19440	3246

Table 8.5: Replication Times in GemStone 2.5

display (or a group of single-unit displays), there are several situations in which a composite display is rendered: e.g., when the top-level display is initially drawn, when a composite display is being scrolled, and when one is involved in a format change.

Smalltalk's algorithm for drawing nested views is first to draw the background of the top view, then its contents, and then repeat this process each of its subviews, applying a display transformation to draw the contents of each view. For deeply nested views, this approach results in refreshing certain areas of the screen many times. However, other implementations of the Smalltalk environment provide improved techniques for rendering deeply nested views. In particular, a double buffering approach updates a bitmap image in memory before rendering the screen image; thus a display's entire screen image would be refreshed only once.

As mentioned earlier, runtime performance was not a major focus of this research while developing the prototype. The time measurements described in this section show that there is much room for improvement in performance. The observations discussed also pinpoint areas of the prototype implementation contributing to poor performance, and indicate that changes in hardware or software platforms can improve the performance considerably. In particular, display generation can be reduced from several minutes down to several seconds.

Chapter Summary

This chapter presented a set of Outlines used to generate displays for the sample database of diets and foods. The displays described by these Outlines exhibit several capabilities that contribute to directness, as discussed in Chapter 2. The ability to define and produce these displays in ODDS shows that the prototype has addressed the basic design objectives that were set forth for the system (in Chapter 3). In the process of creating these displays, several short-comings of the prototype were revealed as well, in the areas of the framework's expressiveness and the usability and performance of the system. These evaluations provide some direction for future work on ODDS.

Chapter 9

Epilogue

This research sought to enhance tool support for developing user interfaces that emphasize directness in working with database objects. A primary objective in the design of ODDS was to support a separation between describing object presentation and describing object behavior and manipulation. This objective provides two main benefits. First, separating the two descriptions promotes an independence that allows each to be modified without heavily affecting the other. Second, the separation promotes display reuse among applications and simplifies experimentation when choosing the displays for an application.

Another primary objective was to provide support for specifying the semantic feedback that is essential to creating directness in object displays. Because implementation of semantic feedback is closely tied to changes made in the displayed objects, its specification poses some difficulty in achieving the separation described above. As a result, many user-interface tools either abandon the separation principle or have limited support for describing semantic feedback. I distinguished three key system features that facilitate a modular, yet comprehensive, description of a display's semantic feedback. The first feature is the use of a behavioral database model for defining object semantics. The second feature is to define the line separating descriptions of display and object behavior such that complex display responses are included in display descriptions. The third feature is to support dynamic representation.

The message paradigm in behavioral database models helps separate display description from definition of application processing. The protocol of an object class provides an abstracted representation of object state and behavior that can be used within display descriptions. At runtime, messages provide a means for the generated displays to invoke object behavior and obtain information on object state without involving the application program. As seen in the sample displays that were created, displays require knowledge of object composition and constraints concerning composition to portray object semantics in a direct manner. Other user-interface development tools have not provided adequate access to that information because the displayed objects are modelled such that the information must be managed by the application program.

A specification framework and runtime-system architecture were designed to include the three features identified above. One main advantage of the framework is that its support for dynamic representation allows the generated dispalys to update display aspects that are typically static once the display has been created. One such aspect is the arrangement and number of display components. Another aspect is the display's focus, i.e., the paths being represented by its subdisplays; changes in display focus provides a means of browsing an object space through a single display. A third aspect made dynamic is the representation of meta-data, such as labels for attribute or relationship names. For example, the title bar in the meal-assignment display names the relationship being presented by the display and can changed at the user's request.

Another advantage of the framework is that it supports declarative description of display behavior within interactors as well as behavior that uses interactors. Many user-interface tools provide a set of interactors whose behavior is not easily modified to suit the particular needs of an application. Such a change often requires familiarity with implementation details of the graphics system underlying the tool. Support for declarative description of interactor behavior helps to ease the task of customizing existing interactors, and thus this support promotes reuse.

Capturing display specifications as data and associating them with object classes provides ODDS with flexibility in several respects. First, applications that use the Outlines may be written using any language interface provided by the DBMS, and the language choice is independent of the implementation language of the runtime system. Second, the concepts and class definitions of the specification framework are not tied to a particular object model or DBMS architecture. Thus, the framework could be used as a basis for developing a display facility similar to ODDS in a variety of OODBMSs.

A prototype of the runtime system was implemented and displays exhibiting various types of semantic feedback were specified and generated. The development of a working prototype verified the major functional components needed to implement the approach of using an OODBMS to supply displays with the semantic information they need. In addition, the prototype showed what runtime data needs to be recorded and what relationships in that data need to be maintained while the displays are active.

Future Directions

This section discusses several improvements to the prototype that would be beneficial for meeting fundamental objective of ODDS, which is to boost productivity in developing object displays that exhibit directness.

First, productivity could be enhanced by extending ODDS with a graphical editing tool that provides several kinds of support for the process of designing and building Outlines. To support the creation of Layout specs, the tool could allow the designer to create the desired image graphically, rather than working in terms of the Layout specs. Ideally, the tool would support either mode for describing image presentation, since some visual features may be described more easily with a spec representation such as the diagram notation used in this dissertation. An Outline-creation tool could also support validation of created specs. The tool could check that the types for element values are correct, or check for other conditions relevant to the correct formation of specs. For example, some action-describing specs should appear only within certain types of Interaction specs. Performing semantic checks before invoking an Outline can often save the effort of trying to discover why a generated display does not function as expected. Another type of support that could be incorporated into the tool is to provide some guidance for the design process, based on the construction model of ODDS.

The specification framework of ODDS is another area where the prototype could be extended. Further research is needed to expand the framework with spec classes that describe additional graphical primitives and layout capabilities. In addition, several extensions for describing interactors and format changes were suggested in Section 8.2. Extending the framework involves adding a subclass to the spec class hierarchy and implementing a corresponding executor class to perform the activites represented by the newly added specs.

In the area of runtime performance, the prototype exhibited considerable delay in display generation of large Outlines and display refresh for deeply nested views. In both areas, the delays were traced to specific implementation techniques and platform-specific features. Thus, a promising direction for future work is to port the prototype to a later release of GemStone (version 2.5 or later) that provides the improvements in object replication and display-refresh capabilities as described in Section 8.2.3.

Bibliography

[Anderson86] T.L. Anderson, E.F. Ecklund, Jr., and D. Maier, "PROTEUS: Objectifying the DBMS User Interface," *Proceedings of the International Workshop on Object-Oriented Database Systems*, ed. D. Dittrich and U. Dayal, Pacific Grove, California, September 1986, pp. 133-145.

[Anson82] E. Anson, "The Device Model of Interaction," *Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 107-114.

[Barth86] P. Barth, "An Object-Oriented Approach to Graphical Interfaces," *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986, pp. 142-172.

[Bass90] L. Bass, E. Hardy, R. Little, and R. Seacord, "Incremental Development of User Interfaces," *Engineering the Human-Computer Interface*, A. Cockton, ed., North Holland, 1990, pp. 21-25.

[Binding87] C. Binding, "The Specification and Implementation of a User Interface Management System Based on a Uniform Output Model," PhD Dissertation, Department of Computer Science, University of Washington, October 1987.

[Bryce86] D. Bryce and R. Hull, "SNAP: A Graphics-Based Schema Manager," *IEEE Conference on Data Engineering*, Los Angeles, California, February 1986, pp. 151-164.

[Butterworth91] P. Butterworth, A. Otis, and J. Stein, "The GemStone Object Database Management System," *Communications of the ACM*, Vol. 34, No. 10, October 1991, pp. 64-77.

[Buxton] W. Buxton, M. Lamb, D. Sherman, and K. Smith, "Towards a Comprehensive User-Interface Management System," *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 35-42.

[Deux91] O. Deux et al., "The O2 System," *Communications of the ACM*, Vol. 34, No.10, October 1991, pp. 34-48.

[Ege87] R. Ege, "Automatic Generation of Interfaces using Constraints," Ph.D. Dissertation, Dept. of Computer Science and Engineering, Oregon Graduate Center, 1987.

[Foley82] J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1982, pp. 218-225.

[Foley86] J. Foley and C. McMath, "Dynamic Process Visualization," *IEEE Computer Graphics and Applications*, March 1986, pp. 16-25.

[Garrett82] M. Garrett and J. Foley, "Graphics Programming Using a Database System with Dependency Declarations," *ACM Transactions on Graphics*, Vol. 1, No. 2, April 1982, pp. 109-128.

[Goldberg83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[Goldman85] K. Goldman, S. Goldman, P.Kanellakis, and S. Zdonik, "ISIS: Interface for a Semantic Information System," *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, Austin, Texas, May 1985, pp. 328-342.

[Green85a] M. Green, "Design Notations and User Interface Management Systems," *User Interface Management Systems*, Eurographics, 1985.

[Green85b] M. Green, "The University of Alberta UIMS," *Computer Graphics*, Vol. 19. No. 3, July, 1985, pp. 205-213.

[Hartson89] R. Hartson and D. Hix, "Human-Computer Interface Development: Concepts and Systems," *ACM Computing Surveys*, Vol. 21, No. 1, March 1989, pp. 5-92.

[Hayes85] P. Hayes, P. Szekely, R. Lerner, "Design Alternatives for User Interface Management Systems Based on Experience with Cousin," *ACM CHI 1985 Conference Proceedings*, April 1985, pp. 169-175.

[Henry88] T. Henry and S. Hudson, "Using Active Data in a UIMS," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, October 1988, pp. 167-178.

[Hill86] R. Hill, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction–The Sassafras UIMS," *ACM Transactions on Graphics*, Vol. 5, No. 3, July 1986, pp. 179-210.

[Hudson87] S. Hudson, "UIMS Support for Direct Manipulation Interfaces," in ACM SIGGRAPH Workshop on Software Tools for User Interface Management, *Computer Graphics*, Vol. 21, No. 2, April, 1987, pp. 120-124.

[HudKing87] S. Hudson and R. King, "Object-Oriented Database Support for Software Environments," *Proceedings of the 1987 ACM SIGMOD Conference on the Management of Data*, May 1987, pp. 491-503.

[Hudson88] S. Hudson and R. King, "Semantic Feedback in the Higgens UIMS," *IEEE Transactions of Software Engineering*, Vol. 14, No. 8, August 1988, pp. 1188-1206.

[Hull87] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Reseach Issues," *ACM Computing Surveys*, Vol. 19, No. 3, September 1987, pp. 201-260.

[Hutchins86] E. Hutchins, J. Hollan, D. Norman, "Direct Manipulation Interfaces," *User Centered System Design*, ed. D. Norman and S. Draper, Lawrence Erlbaum Associates, Inc., 1986, pp. 87-123.

[Ingalls88] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, K. Doyle, "Fabrik: A Visual Programming Environment," *OOPSLA '88 Proceedings*, September 1988, pp. 176-190.

[Jacob86] R. J. K. Jacob, "A Specification Language for Direct Manipulation Interfaces," *ACM Transactions on Graphics*, Vol. 5, No. 4, October, 1986, pp. 283-317.

[Kasik82] D. Kasik, "A User-Interface Management System," *Computer Graphics*, July 1982, pp. 99-106.

[King89] R. King and M. Novack, "FaceKit: A Database Interface Design Toolkit," *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, August 1989, pp. 115-123.

[Laenens89] E. Laenens, F.Staes and D. Vermeir, "Browsing a la carte in Object-Oriented Databases," *The Computer Journal*, Vol. 32, No. 4, August 1989, pp. 333-340.

[LaLonde91] W. LaLonde and J. Pugh, *Inside Smalltalk Vol. 2*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Larson86] J. Larson, "A Visual Approach to Browsing in a Database Environment," *IEEE Computer*, Vol. 19, No. 6, June 1986, pp. 62-71.

[Leong89] M.K. Leong, S. Sam, and D. Narasimhalu, "Towards a Visual Language for an Object-Oriented Multi-Media Database System," *Visual Database Systems*, ed. T. L. Kunii, Elsevier Science Publishers, 1989, pp. 465-496.

[Lieberman86] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented System," *OOPSLA '86 Proceedings*, September 1986, pp. 214-223.

[Linton89] M. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, Vol. 22, No. 2, February 1989, p. 8-22.

[McCarthy89] D. McCarthy and U. Dayal, "The Architecture of an Active Data Base Management System," *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, June 1989, pp. 215-224.

[Maier86] D. Maier, P. Nordquist and M. Grossman, "Displaying Database Objects," *First International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

[Maier87] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," *Research Directions in Object-Oriented Programming*, B. Shriver ed., MIT Press, Cambridge, 1987, pp. 355-392.

[Myers86] B. Myers and W. Buxton, "Creating Highly-Interactive and Graphical User Interfaces by Demonstration," *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 249-258.

[Myers89] B. Myers, "User-Interface Tools: Introduction and Survey," *IEEE Software*, Vol. 6, No. 1, January 1989, pp. 15-23.

[Myers90] B. Myers, ed., "The Garnet Compendium: Collected Papers, 1989-1990," CMU-CS-90-154, Carnegie Mellon University, August 1990.

[Olsen83] D. R. Olsen and E. Dempsey, "Syngraph: A Graphical User-Interface Generator," *Computer Graphics*, July 1983, pp. 43-50.

[Olsen86] D. R. Olsen, "MIKE: The Menu Interaction Kontrol Environment," *ACM Transactions on Graphics*, Vol. 5, No. 4, October 1986, pp. 318-344.

[ParcPlace] ParcPlace Systems, *VisualWorks Users Guide, Version 1.0.*

[Pilote83] M. Pilote, "A Data Modeling Approach to Simplify the Design of User Interfaces," *Ninth International Conference on Very Large Data Bases*, ed. M. Schkolnick and C. Thanos, Florence, Italy, October, 1983, pp. 290-299.

[Pfaff85] G. Pfaff, ed., *User Interface Management Systems*, Springer-Verlag, Berlin, 1985.

[Plateau89] D. Plateau, R. Cazalens, and B. Poyet, "A Customizable Abstract I/O Server for Complex Object Edition," Altair Technical Report 28-89, March 1989.

[Rosenberg88] J Rosenberg et al., "UIMS: Threat or Menace," *ACM CHI 1988 Conference Proceedings*, May 1988, pp. 197-200.

[Rowe82] L. Rowe, "A Form Application Development System," *Proceedings of the 1982 ACM SIGMOD International Conference on the Management of Data*, Orlando, Florida, 1982, pp. 28-38.

[Rowe91] L. Rowe et al., "The PICASSO Application Framework," *Proceedings 1991 ACM Symposium on User Interface Software and Technology*, Hilton Head, South Carolina, November 1991, pp. 95-106.

[Rowe92] L. Rowe and K. Shoens, "A Retrospective on Database Application Development', *SIGMOD Record*, Vol. 21, No. 1, March 1992, pp. 5-10.

[Schmidt90] D. Schmidt, "From Object-Oriented Database Systems to High Productivity Software Development Environments," OGI Technical Report 91-104, July 1990.

[Schulert85] A. Schulert, G. Rogers, and J. Hamilton, "ADM: A Dialogue Manager," *ACM CHI 1985 Proceedings*, New York, April 1985, pp. 177-183.

[Sibert86] J. Sibert, W. Hurley, and T. Blesser, "An Object-Oriented User Interface Management System," *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 259-268.

[Schmucker86] K. J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hasbrook Heights, New Jersey, Hayden Book Company, 1986.

[Stonebraker86] M. Stonebraker and L. Rowe, "The Design of Postgres," *Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data*, Washington, D.C., May 1986, pp. 340-355.

[VanderZanden89] B. Vander Zanden, *Incremental Constraint Satisfaction and its Application to Graphical Interfaces*, Ph.D. Thesis, TR 88-941, Dept. of Computer Science, Cornell University, 1989.

[Webster89] B. Webster, *The NeXT Book*, Addison-Wesley, 1989.

[Weinand89] A. Weinand, E. Gamma, and R. Marty, "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework," *Structured Programming*, Vol. 10, No.2, 1989, pp. 1-25.

[Zdonik90] S. Zdonik and D. Maier, "Fundamentals of Object-Oriented Databases," *Readings in Object-Oriented Database Systems*, Morgan and Kaufman Publishers, 1990, pp. 1-32.

[Zhu89] J. Zhu, *Model, Language and Implementation Aspects of a Logic-Based Object-Oriented Database System*, Ph.D. Dissertation, Dept. of Computer Science and Engineering, Oregon Graduate Institute, July 1989.

Biographical Note

The author was born Belinda Paz Capistrano Buenafe on Okinawa, Japan in 1964. In 1985, she earned a Bachelor of Science in Electrical Engineering at the University of Portland in Oregon. Several months later, she began her graduate studies in computer science at the Oregon Graduate Center (now OGI). There she became involved in research on combining the latest technology in the areas of user-interface construction tools and object-oriented databases. While completing the requirements for her Ph.D., she took a position within an engineering group at ParcPlace Systems, where she continues to work on object-oriented technology and its practical application.