# INTERACTIVE DISPLAY GENERATION IN SMALLTALK

Peter R. Nordquist
B.Mus.Ed., George Fox College, 1979
B.Mth.Ed., George Fox College, 1979

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

March, 1985

The thesis "Interactive Display Generation in Smalltalk" by Peter R. Nordquist has been examined and approved by the following Examination Committee:

David Maier
Associate Professor
Dept. of Computer Science and Engineering
Thesis Research Advisor

Robert G. Babb II
Assistant Professor
Dept. of Computer Science and Engineering

Richard Hamlet
Professor
Dept. of Computer Science and Engineering

Ralph L. London
Adjunct Professor
Computer Research Laboratory
Tektronix, Inc.

## Dedication

To Chris, my wife and best friend, without whom this thesis would not have been possible.

## Acknowledgements

I would like to thank my advisor, Dr. David Maier, for his direction, understanding, and support. I would also like to thank Mark Ballard for his work on the software used to print the bit-maps in this thesis, as well as for many fruitful discussions about Smalltalk. I am grateful to Harry Porter for the initial writing of the program for printing bit-maps, and to the people at the Computer Research Laboratory, Tektronix, Inc. for supplying the magnolia workstation and a running Smalltalk system.

I am most indebted to my parents for their encouragement and faith in me throughout my stay at O.G.C., and also to my wife's parents for their support.

# Table of Contents

# List of Figures

# Abstract

INTERACTIVE DISPLAY GENERATION IN SMALLTALK

Peter R. Nordquist, M.S.

Oregon Graduate Center, 1985

Supervising Professor: David Maier

Smalltalk is an object oriented programming language designed to be run on a workstation with a dedicated processor, bit-mapped display, and pointing device. Smalltalk environments are highly interactive with multiple overlapping windows appearing on the screen simultaneously. These windows or Interactive Displays (IDs) both display data and monitor the mouse and keyboard for user input. They accomplish these actions by making use of multiple Model-View-Controller (MVC) triples. IDs are central to Smalltalk, yet constructing them is an unduly complicated process.

SIG, the system described herein, eases the process of creating IDs in Smalltalk by introducing display types and abstract views. A display type is a specification for an ID. Display types provide a mapping from a user's data structure to an ID on that data structure. Display types are "interpreted" by abstract views. Together, display types and abstract views allow IDs to be generated from specifications. New MVC triples can be added (and removed) from a generated ID dynamically and conditionally, based on the current status of the user's data structure. In addition, display types may reference one another to compose IDs.

# 1. Introduction

Computing has been plagued from its outset by an inability on the part of people to *see* what it is that is inside the computer, i.e., just what data is actually stored there. Thus user interfaces and their design have been and continue to be of central importance in computer science. In particular, this thesis is concerned with user interfaces for editing data structures and how such interfaces can be specified. The editing system described herein operates on data structures in Smalltalk[1] and makes use of a high resolution bit-mapped display with pointing device.

## 1.1. Interactive Displays

Originally, data structures were viewed and described as text.[2] Interactive editors dealt with all objects as if they were text, so all objects had to be reduced to a textual representation to be edited. Gradually, as technology allowed, structured data began to be represented as such, making the creation of interactive editors a more difficult task. Meyrowitz and Van Dam have given a thorough summary of interactive editing systems dating from the days of input via card decks through today's high resolution, bit-mapped displays with graphic input devices on personal workstations [Meyr82].

The interactive editors in this treatise are called *interactive displays*, hereinafter written ID. For the purpose of this document, an ID is a two-dimensional display of a data structure with which a user can interact. An ID receives and executes user commands until told to close by the user. Positioning the cursor on this display creates an association with a particular portion of the data structure being displayed. Entered data then affects that particular part of the

---

[1]Smalltalk was developed by Xerox Corporation at Xerox Palo Alto Research Center [Gold83]. The system described herein was developed on a Tektronix Magnolia workstation running Smalltalk-80 version 2.

[2]Numbers are considered text for the purposes of this discussion.

data structure being displayed. The data structures for the IDs built by the system described herein may be recursively defined and of arbitrary complexity. The display changes dynamically to reflect the current state of the data structure.

Foley and Van Dam have given guidelines for how to build an interactive graphics system [Fole82], including how a data structure can be broken into segments that are then composed to form an image on the screen. These segments are expressed graphically in their own coordinate system, which is then translated to the coordinate system of the screen through a series of viewing transformations. Fortunately, this work has already been done in the viewing mechanism of Smalltalk.

### 1.2. Databases

Some work on graphical displays of data structures has been done for databases. A representative system is the Spatial Data Management System [Hero80]. At the user's disposal are three screens. One provides a world-view of the database, i.e., a coarse index of the whole system. On the other two, the user can obtain an exploded view of a portion that is highlighted on the world-view screen. The data, for the most part, is represented as icons annotated with written information for numeric and textual data.

Roger King has written an object oriented database management system called *Sembase* [King84]. Objects in this system are instances of types. Types have attributes, and each object maintains values for the attributes defined by that object's type. The system runs on a high resolution bit-mapped display with a mouse. The system interface makes use of a fixed screen format to display any and all data structures and type-attribute information. There is a fixed set of operators the user can invoke, using the mouse, to manipulate the displayed data. Thus, while this system does display data structures and allow the user to perform operations on those structures, the display is static in that the screen format and mouse menu operators are system defined and can not be specified by the user.

### 1.3. Program Visualization

Program visualization systems are designed to aid in the building and debugging of programs by displaying data structures associated with programs, including source code, while the program is running. The data structures displayed are in some way tied to the running program. Steven Reiss at Brown University has an excellent example of such a system, called PECAN [Reis83]. It offers multiple, concurrent program views, and has a syntax directed editor. The text presented in this editor can be edited as text or traversed as an abstract syntax tree; the mouse or keyboard can be used to effect this traversal. Whenever a change is made in the source program, all other views are updated to reflect this change.

Program visualization systems are important as related work because they have multiple views that change dynamically when the structure they are displaying changes. They are different in that they do not operate on arbitrary, user-defined, data structures. Another programming system with visualization, PV, is described in [Hero82] and [Kram83]. Fischer has a similar system for documenting Lisp programs [Fisc84]. The thrust of this system is documentation rather than actual program visualization. The major contribution of this system is that it allows users to specify which parts of the documentation they want to see by means of a filter. This idea is also seen in the change management browser in Smalltalk v.2 [Gold84].

### 1.4. Algorithm Animation

Algorithm animation is concerned with showing data structures change over time as an algorithm acts on them. Algorithm animation differs from program visualization for the most part in the kind of data structures that are displayed. Both kinds of systems are designed to visually display programs as they run, but differ in the kind of data they display and in the way they display that data. Almost exclusively, views in program visualization systems are tied to the source code in some way. Typically, one view will show source code statements highlighted as they are executed, another, a representation of the syntax tree for this source code, and

another a view of the activation record stack. All of these views are related to the code itself in a straightforward way. Algorithm animation, on the other hand, seeks to graphically explain the algorithm that is implemented by the code. Rarely is the code that implements an algorithm the object of a graphic view as it is in program visualization. A good example of algorithm animation is found in the film *Sorting Out Sorting* by Ronald Baecker [Baec81]. This film represents differing values by variable length rods or sticks that are dynamically rearranged as a sorting algorithm executes. Typically, the small sticks are moved to the left or upper part of a graphic view and the large ones to the right or lower part of this view. The order in which the sticks are moved depends on the algorithm being animated. Animating algorithms generally requires multiple views that are dynamically updated as data structures change. A representative system is Brown's BALSA [Brow84]. Ralph London, Robert Duisberg, and Richard Wagner have recently produced an algorithm animation system in Smalltalk, and in doing so, have run up against some of the problems of the model-view-controller paradigm described later in this document [Lond84]. Algorithm animation provides innovative, well integrated views on the data structures that are predefined by the algorithm being animated.

### 1.5. Data Structure Display

Brad Myers developed a system, *INCENSE*, at Xerox Parc for the language *Mesa* [Myer83]. Mesa is strongly typed. Incense has display primitives for the basic types, and displays can be created for records and arrays that are built using these basic types. Every item to be displayed has an Artist, a set of display procedures. When called (from Mesa), an Artist is given the position and size of the area on the screen in which the item the Artist is responsible for displaying is to be displayed. It is the Artist's job to fit the data to be displayed into the area it has been given. Incense was designed as an aid for interactive debugging. It does support the display of arbitrarily complex, including recursive, structures built from the basic types, but does not allow interactive editing on those structures.

### 1.6. Smalltalk Inspectors

Smalltalk, as released by Xerox, has a facility for viewing and editing data structures. Data editors in Smalltalk are called inspectors. Inspectors do allow editing of arbitrary, user defined data structures, but they do so using a fixed screen format. Figure 1.1 shows an inspector on an Smalltalk object that is a rectangle. All inspectors have two views, regardless of the structure of the data being viewed. All objects are represented in the same way within these two views. Inspectors are further explained in chapter 3.

### 1.7. Other Related Work

Two other topics, while not directly related, should be mentioned here. Several people have addressed the problem of designing specification languages for drawing pictures [Hend82], [Pere83], [Wyk82], [Egge83]. Also, document preparation systems have to deal with displaying data on paper. These systems solve this problem by supplying a specification language for formatting data. This data consists of text, pictures, equations and tables. The format for printing this data is specified using the specification language, but the exact placement of data items on the page must be determined by the system [Furu82], [Kimu83], [Knut79].

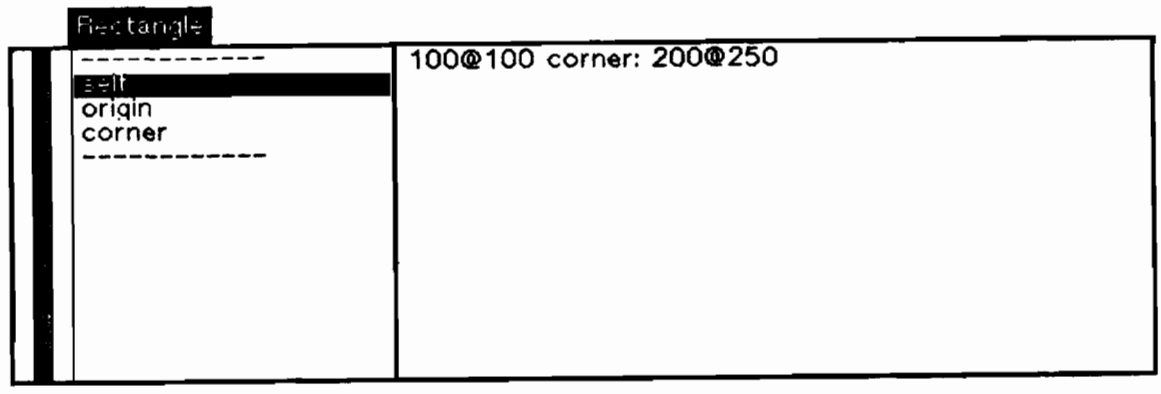


Figure 1.1 A Smalltalk Inspector

## 1.8. Specification

This thesis describes a means of specifying IDs. Specification languages have been important to computing since its inception; indeed, all programming is specification at some level. Specification languages are used for a variety of purposes. General purpose programming languages specify a sequence of machine instructions; database definition languages specify the type of data to be stored in a database; and report generation languages specify the way data is to be analyzed and presented. Another important use of specification is found in parser generation. Parser generators input a grammar for a programming language. This grammar delineates the constraints that a program must meet in order to be syntactically correct. The parser generator then produces a parser which analyzes source programs according to these constraints. The system described herein does something similar, taking a specification for an ID on a data structure as input and producing an ID on that data structure.

In the past, most editors were text editors. It was assumed that the text was a long string of characters, and any structure imposed on this text was maintained by the user so was of no consequence to the text editor itself. This assumption made it unnecessary to have editor generators, for all editors carried out essentially the same operations, leaving no specification to be done. Recently, work has been done in document preparation to add structure to documents. Kimura and Shaw have proposed a model for documents [Kimu83]. This model is composed of document objects that are either abstract or concrete, ordered or unordered, and composed in a hierarchy. These document objects, much like Smalltalk objects, are instances of classes. These classes can then be specified, and document editors produced that create documents structured according to the specifications.

As technology allowed, graphics editors came into being. Graphics editors that treat the image produced as one that has structure are candidates for specification. Constraints on the structure of the images produced by an editor are given in the specification for that editor. In the examples in the next chapter, the data objects being edited have structure and the IDs

produced for these data structures consist of boxes containing text that are composed in such a way as to reflect the structure of the underlying data. In this ID, we want to edit both the text in the boxes and the placement of the boxes themselves.

### 1.9. Overview

The problem addressed in this work is providing high-level specifications that can be used to create IDs on data structures.

#### 1.9.1. Requirements

The solution must meet the following minimum requirements.

(1) High-level specifications must be as declarative as possible. The major role of these specifications is to describe *what* the appearance and behavior of IDs are, not *how* this appearance and behavior are provided.

(2) IDs generated from these high-level specifications must dynamically reflect the structure of the data object displayed. The structure of an ID is based on the structure of the data it is displaying. Thus, when the structure of this data changes, the structure of the ID must change accordingly. For example, an ID displaying an array of objects may allot an amount of screen area to each element in the array. When elements are added or deleted, the screen area needs to be reallocated to effectively display the new state of the array.

(3) IDs must be able to display data structures with arbitrary levels of structure. Binary trees are examples of such structures. When nodes are added or deleted from a binary tree, the depth of the tree may change. The structure of an ID displaying such a tree must change in response to the change in the tree.

(4) The high-level specifications must provide a framework or design methodology for building up IDs. It must be possible to specify individual pieces of an ID separately, as modules.

### 1.9.2. Vehicle

The system developed to solve this problem uses Smalltalk-80 version 2, in particular, the model-view-controller (MVC) paradigm. The MVC paradigm is used to create all of the IDs supplied with the Smalltalk system. In the MVC paradigm, an ID is a hierarchy of MVC triples. Each MVC triple consists of three distinct objects: a model, a view, and a controller. In the MVC paradigm, the model stores the data to be displayed, the view displays this data, and the controller functions as an editor by taking command input from the user. The MVC paradigm by itself falls short of meeting the requirements above for the following reasons.

(1) Creating an ID with just the MVC paradigm is a procedural, not a declarative, process. In general, new Smalltalk code must be written for the models, views, and controllers that make up an ID. There is no declarative specification mechanism.

(2) IDs created with just the MVC paradigm usually have a fixed format. This format does not change to reflect changes in the structure of the data displayed. Most IDs are limited to a two-level MVC triple hierarchy. Thus, these IDs do not represent recursive or iterative data structures well.

(3) It is difficult to create modular IDs with the MVC paradigm. Most IDs consist of a mass of interrelated pieces that are not easily separated. In addition, the framework provided by the MVC paradigm is too open ended. Design questions concerning which object (the model, view, or controller) should implement which function are difficult to resolve. In the system-supplied IDs, often both the view and controller store data and the controller often does the majority of the display updating.

### 1.9.3. Solution

This thesis describes the SIG system, which addresses the problem of providing high-level specifications from which IDs are created. SIG introduces two new concepts: *display types* and *abstract views*. Display types allow design decisions to be "factored out" into specifications

separate from the IDs themselves. A display type is a template that, when filled in, specifies the appearance and update behavior of an ID. Abstract views interpret display types and adapt an ID to changes in the structure of the data displayed.

Display types are associated with Smalltalk classes. One class may have multiple display types, each producing different IDs for objects that are instances of that class. Creating multiple display types for a class allows users to represent instances of the class in a variety of ways. Each display type has a list of *recipes*. Recipes provide the means for an ID changing in response to changes in the data structure displayed by that ID. Each recipe has a condition and a list of *view specifications*. The conditions are applied to the data structure being displayed to determine which recipe should be invoked. Each view specification describes the behavior and appearance of a piece of the ID. View specifications thus allow the description of an ID to be broken into manageable pieces.

Display type editors are provided to help users deal with the complexity inherent in constructing display types. A display type editor keeps the recipes and view specifications associated with each display type in order. A display type editor enforces a helpful regimentation when a user fills in a display type. When a user is editing a particular part of a display type, e.g., a recipe, only the view specifications for that specific recipe are available for editing. In addition, templates are provided for each part of a view specification to aid the user in correctly specifying ID behavior and appearance.

SIG also provides a set of primitive views. These primitive views cover most of the common behaviors exhibited by IDs on simple data objects. When a user indicates, in a view specification, that a particular part of an ID is to use a primitive view, a set of behaviors is provided automatically. The user has then only to supplement these basic behaviors to describe totally that part of the ID. For example, an ID on a binary tree needs to have a view in which the user can edit the text constituting the label for a node, and SIG provides primitive views for displaying and editing text.

Abstract views provide the backbone for IDs generated from display types. IDs are composed of primitive views and abstract views. Abstract views have one or more subviews. Each of these subviews is either another abstract view or a primitive view. The number, type, and placement of subviews depends on the structure of the data item displayed. Abstract views are responsible for accurately representing the structure of the data item they are displaying. Thus, an abstract view holds a display type that describes this structure. When the structure of the data item changes, the abstract view consults the list of recipes that comprise the display type to determine which applies to the new state of the data item. The abstract view then adds new subviews as specified by the chosen recipe. For example, in an ID of a binary tree, one abstract view is responsible for displaying one node of the tree. An abstract view displaying a node with no children has one subview, a primitive view, to display the label of the node. When this node acquires a child, the abstract view, after consulting the recipes of its display type, re-adds the subview displaying the node's label, and adds a new subview (another abstract view) to display the new child.

### 1.9.4. Adequacy of Solution

Together, display types and abstract views meet the requirements imposed on the solution, given above.

(1) Display types are a high-level specification for IDs on data structures. Display types specify, declaratively for the most part, the parameters that are used to create an ID. Display types also list conditions that are used to determine how an ID is to reflect the various states of a data structure.

(2) SIG-generated IDs change dynamically with changes in the structure of the data displayed. These changes are accomplished by abstract views that use the state of data structures to select a particular recipe from a display type. The list of view specifications associated with each recipe designates the number, type, and placement of the subviews of

an abstract view. Recursive data structures are accurately and dynamically portrayed because abstract views hold display types containing recipes that call for new abstract views and delete and re-add subviews while an ID is in progress. Iterative structures can likewise be portrayed by providing a recipe for each number of iterations, though specifying display types for these structures is somewhat more difficult than specifying the display of recursive structures.

(3) Display types provide modularity at two different levels. View specifications specify a single MVC triple that is part of a larger ID. Display types specify sets of such triples. Display types can reference other display types to create a hierarchy of specifications that together compose an entire ID. Display type editors provide a design methodology for the creation of IDs. An ID is created by filling in a display type(s) and writing Smalltalk code to allow data objects to respond to the messages sent to them by the views specified in view specifications. Display type editors aid the user in correctly filling in display types. Primitive views also aid the process by providing the user with basic building blocks for an ID. This design methodology lends uniformity to the process of creating IDs in Smalltalk. The user is insulated from most of the complexity of the MVC paradigm because display types are specifications for and separate from the MVC triples that actually realize IDs. The Smalltalk code that the user does write is all associated with the data objects (models) displayed in an ID, not the views and controllers implementing that ID.

# 2. A Sample Interactive Display

Figure 2.1 shows an ID on a data structure that is a tree, each node of which has an arbitrary number of children. This data structure represents a Prolog literal. Each node in the tree represents the functor for some Prolog literal, and the children of this node, the arguments of the functor. The requirements for this ID are that it faithfully represent the structure of the tree, display the functor at each node, allow editing of the functor at each node, allow nodes to be added and subtrees deleted, and update the linear representation of the tree whenever any of these edits are performed.

An entire Prolog clause displayed in this ID can be thought of as a single Prolog literal with functor, ':-', and an ordered list of arguments. The first argument in the list represents the head of the clause and the following arguments, the body. Each of these arguments is also

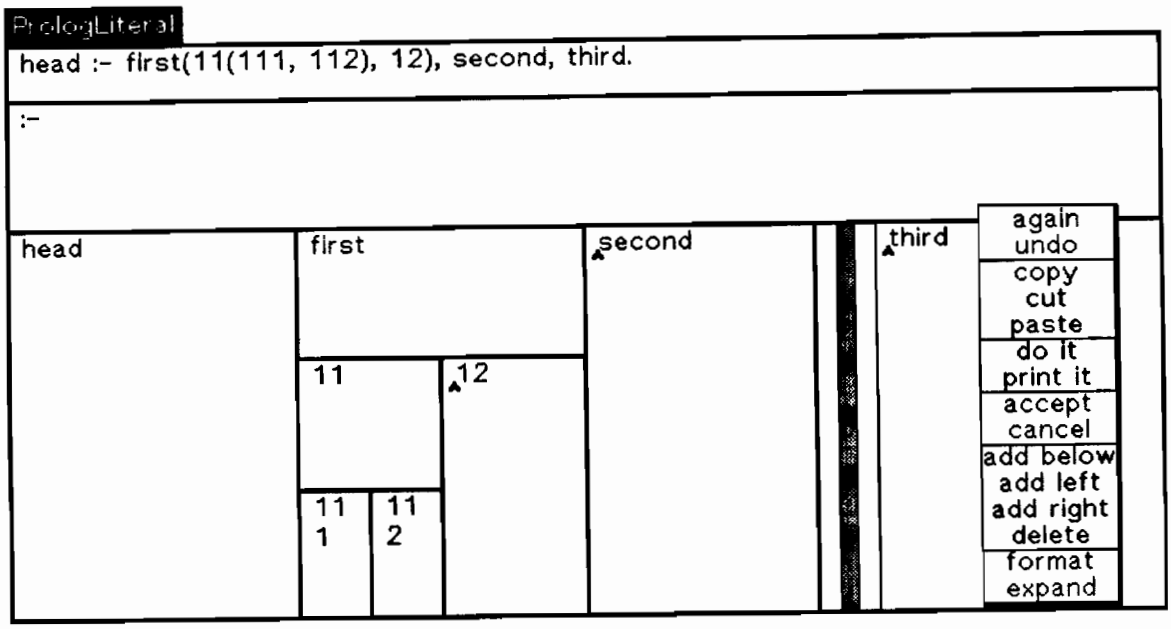


Figure 2.1 Example Prolog Literal

a literal, an instance of the Smalltalk class PrologLiteral,[1] thus a clause is a tree in which every node represents the functor of some Prolog literal. Figure 2.2 shows the general scheme used to display a Prolog literal in Figure 2.1. For any given Prolog literal, the arguments appear in order, below the functor. The view for the functor spans the width of the screen area allotted to display the Prolog literal. For example, the view containing the text *first* in Figure 2.1 spans the width of the views displaying the texts *11* and *12*, the texts of the functors of the arguments of functor *first*.

The uppermost view in Figure 2.1 shows a textual representation of the entire Prolog clause. In this view, the functor, :-, at the root of the tree that represents the entire clause is displayed following its first argument, *head*. The functor :- is then displayed followed by the rest of its arguments, *first(11(111, 112), 12), second, third*. This is the way Prolog clauses normally read. All other Prolog literals in the uppermost view, e.g., *first(11(111, 112), 12)*, are displayed in the following format: functor first followed by the argument list in parentheses. The next lower view contains the text :-. This view spans the width of the whole figure because

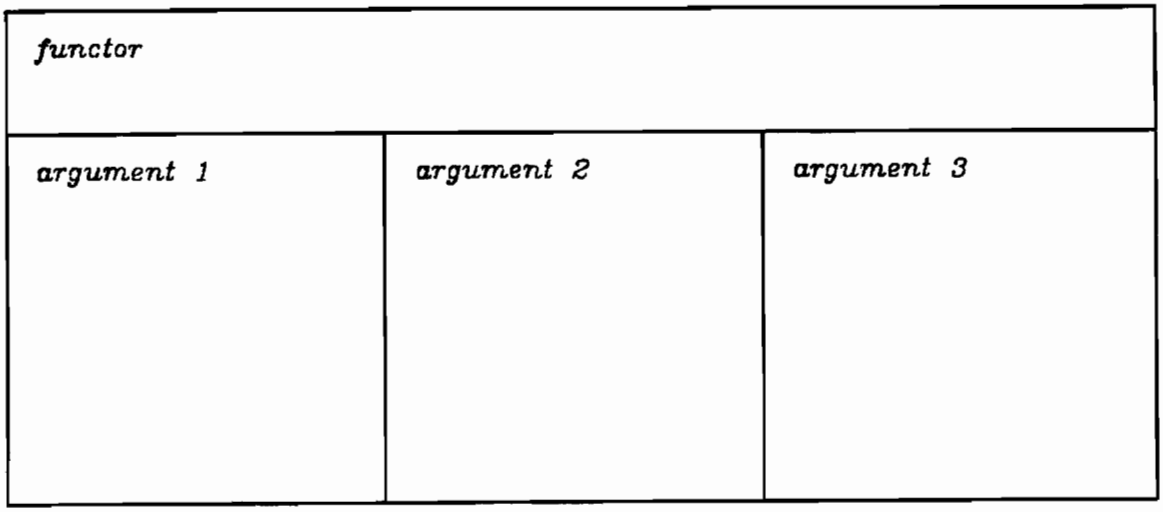


Figure 2.2 Viewing Scheme for a Prolog Literal

[1]Smalltalk class names must start with a capital letter and be only one word long. The convention for "multiple" word names is to capitalize the first letter of each word.

it is showing the functor :-; this is the functor for the Prolog literal that embodies the entire clause. The arguments of :- are displayed below it, in order, from left to right. Observe how the scheme shown in Figure 2.2 is repeated recursively for each Prolog literal in the clause. Views for functors that have no arguments extend to the bottom of the ID. Each view, except the one showing the linear representation, displays the functor for a particular Prolog literal. By moving the pointing device inside a view, one can edit the functor shown in that view. Figure 2.3 shows functor *third* changed to *xxxx*. After accomplishing this edit, the uppermost view is updated to show the current state of the clause, as shown in the figure.

In addition to editing the functors of existing Prolog literals, we need to be able to add and delete Prolog literals. When a Prolog literal is added or deleted, the structure of the clause changes and this change must be reflected in the display. This structural change means views must be added or deleted and portions of the display redrawn. Figure 2.4 shows the adding of an argument, a Prolog literal with functor *newFunctor* and no arguments, to the functor *xxxx*.

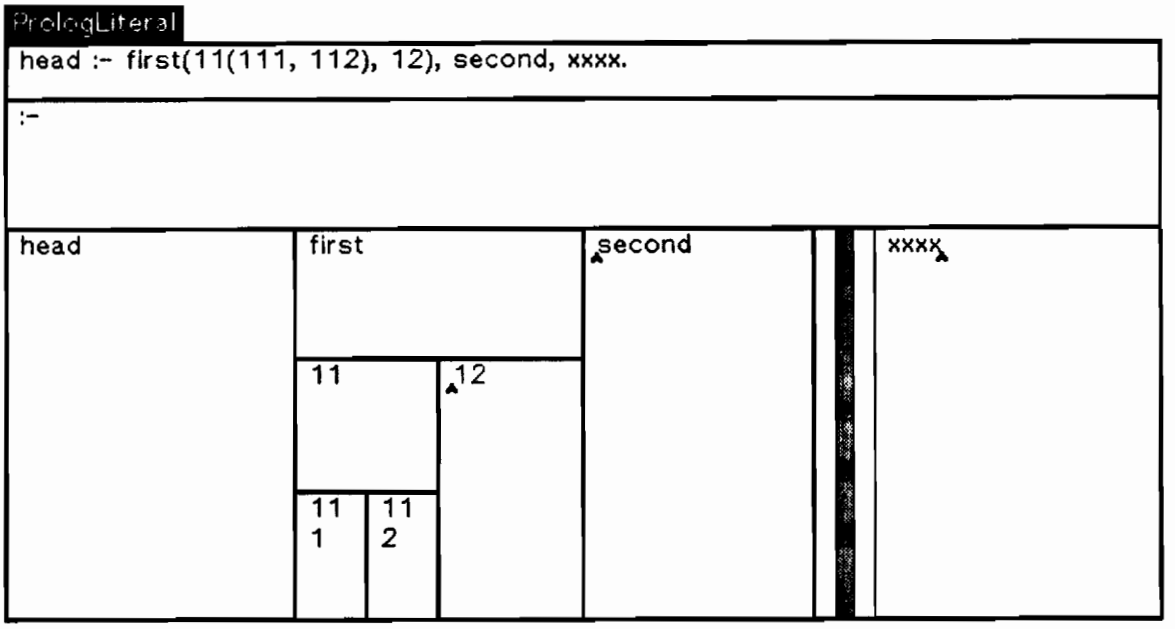


Figure 2.3 Editing a Functor

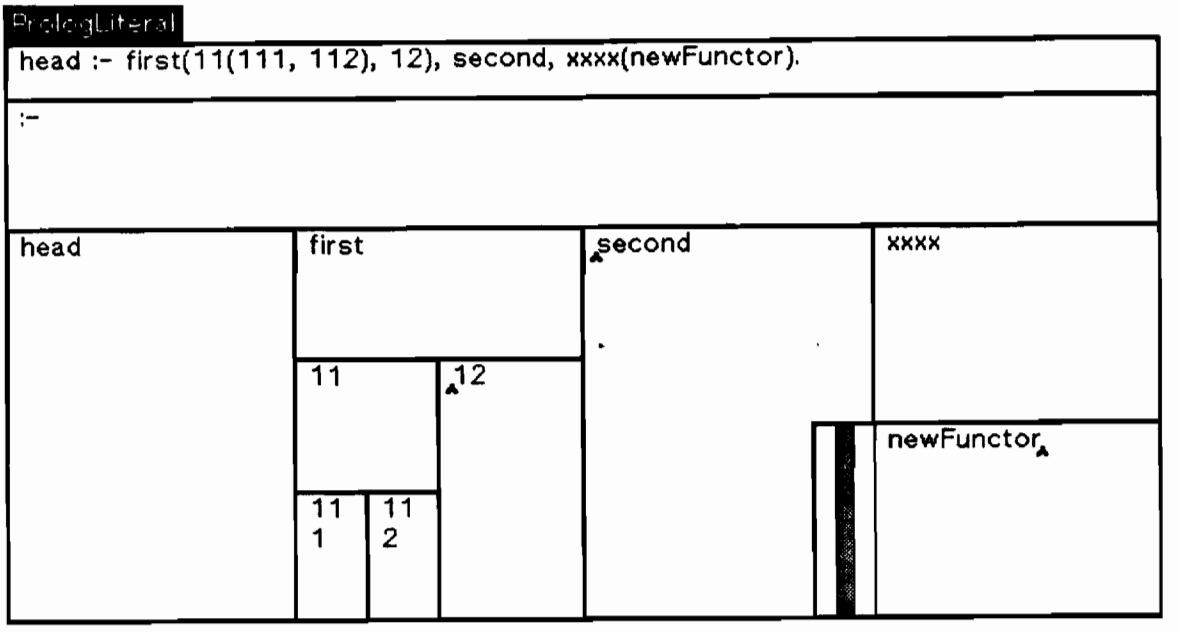


Figure 2.4 Adding a Prolog Literal

Adding this argument necessitated splitting the area formerly occupied by *xxxx* into two views. Notice that the uppermost view showing the linear representation is also updated. Since a Prolog literal encompasses not just one node but an entire subtree, when a Prolog literal is deleted, its entire subtree should disappear. Figure 2.5 shows the deletion of the literal whose functor is *11*. Again, views had to be reallocated and the linear representation updated.

A Prolog literal was chosen as a sample data structure because it exhibits most of the features that make a data structure difficult to display. It is recursive, so there is no a priori bound on the number of functors to be displayed. Each functor may have any number of arguments, so the number and size of views needed for any given functor can not be preset. In addition, views must be dynamically added and destroyed when functors are added and deleted, so the ID must have the ability to analyze the data structure and use this analysis to allocate display space. Of course when space is allocated dynamically, the ID must have some way of dealing with the eventuality of running out of space.

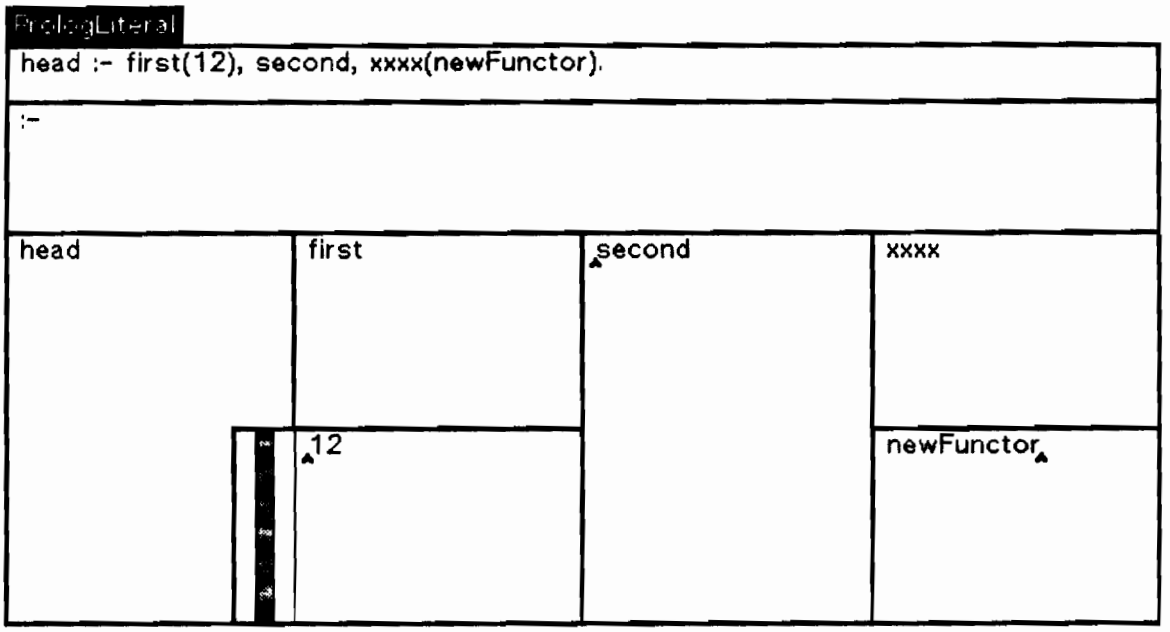


Figure 2.5 Deleting a Prolog Literal

# 3. Smalltalk and the Model-View-Controller Paradigm

Smalltalk is an object-oriented programming language. It has strong support for graphics and is meant to run on a workstation with a high-resolution, bit-mapped display. In Smalltalk, objects communicate with each other by passing *messages*, which are implemented by *methods*. Chapter 2 introduced an ID on a PrologLiteral. PrologLiterals are Smalltalk objects that respond to the message *addBelow*. Whenever the user selects menu option *add below*, (see Figure 2.1) a PrologLiteral receives the message *addBelow*. The method implementing the message *addBelow* follows.

```
addBelow
    "1) create a new PL
    2) add it below me
    "

    self changeRequest
    ifFalse: "Someone doesn't want me to do this add"
        [↑self].
    self addBelow: PrologLiteral new.
    self topNode changed: #clauseAsText
```

The character '.' is the statement separator in Smalltalk. There are three kinds of message selectors: unary, binary, and keyword. Unary selectors are simply identifiers; *changeRequest* above is a unary selector. Binary selectors are infix operators. Arithmetic operators are binary selectors. Keyword selectors consist of a word or set of words each followed by a colon and an argument; *addBelow:* is a keyword selector in the example above, and the result of the unary message *new* sent to class PrologLiteral is its argument. The method shown above delineates the actions that are taken whenever a PrologLiteral receives the message *addBelow*. First of all, the PrologLiteral sends itself the message *changeRequest*. This message returns true if it is permissible for the PrologLiteral to change. If the message *changeRequest* returns false, this method terminates and returns the receiver (the PrologLiteral that received the message

*addBelow*), otherwise, the PrologLiteral sends itself the message *addBelow: PrologLiteral new*. This message creates a new PrologLiteral that is then added directly below the receiver in the tree of PrologLiterals of which the receiver is a part. After this add has been accomplished, the PrologLiteral sends itself the message *topNode*. This message returns the PrologLiteral embodying the entire Prolog clause displayed by the ID. This PrologLiteral (the top node) is then sent the message *changed: #clauseAsText*[1] which updates the uppermost view in the ID. Recall this view is the one displaying the linear representation of the entire clause, see Figure 2.1.

Smalltalk objects are grouped into *classes*. For instance, there is the class of all rectangle objects. These objects are *instances* of the class *rectangle*. For each class, there is a class describing object that describes the structure (instance variables) that the instances of the class have and contains the methods for the messages to which the instances can respond. Classes may have *subclasses*. A subclass is a specialization of a class. A subclass inherits both the structure and methods of its *superclass*, the class of which it is a subclass. For example, suppose we define a class of persons. Instances of the *persons* class respond to the messages *name* and *age* which return to the sender the name and age, respectively, of the person object to which the message was sent. We could then define a class, *employees*. The class of employees is a subclass of the class of persons. Instances of the class of employees respond to the messages *name* and *age* by using the same methods that instances of the persons class use. However, employee objects also respond to the message *salary*, whereas person objects do not.

### 3.1. System Supplied IDs in Smalltalk

Smalltalk abounds with IDs on data structures, indeed, the whole user interface is built around IDs. System browsers are probably the most common ID. They are used to maintain class definitions and the methods associated with each class. Figure 3.1 shows a system browser

[1]The construct *#clauseAsText* represents a literal message selector. Message selectors are instances of class Symbol. A constant Symbol is represented in Smalltalk by preceding the Symbol itself with #. Typically, the Symbol sent as the argument of the *changed:* message is a constant. In this case, *clauseAsText* is a Symbol constant, i.e., a specific, user defined Symbol, and the preceding # denotes this fact.

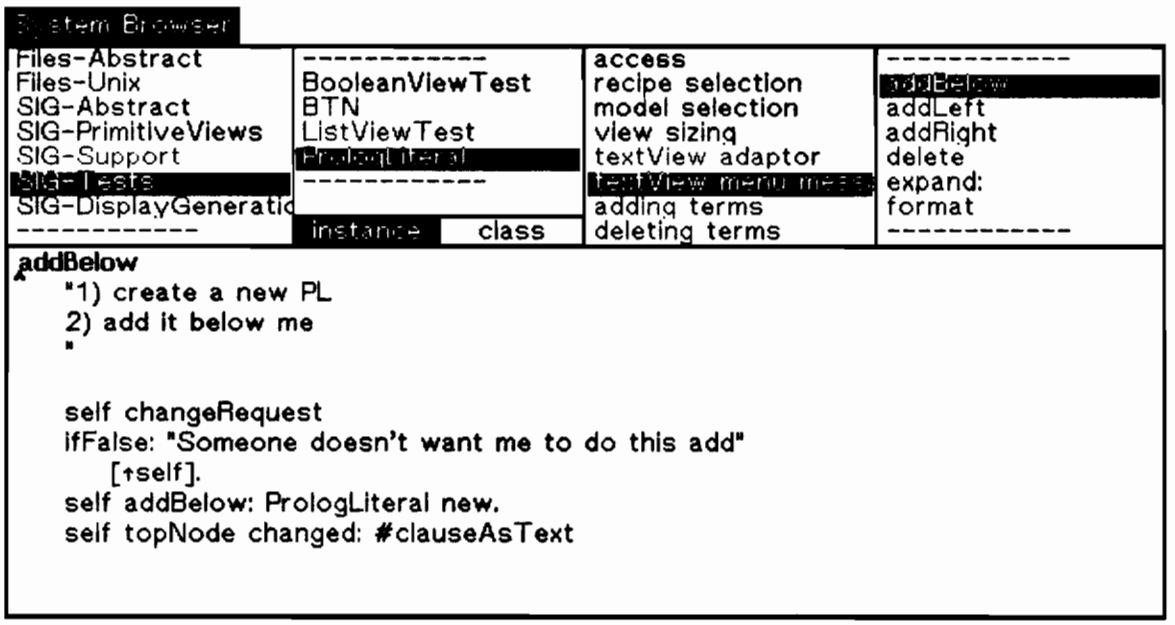


Figure 3.1 A System Browser

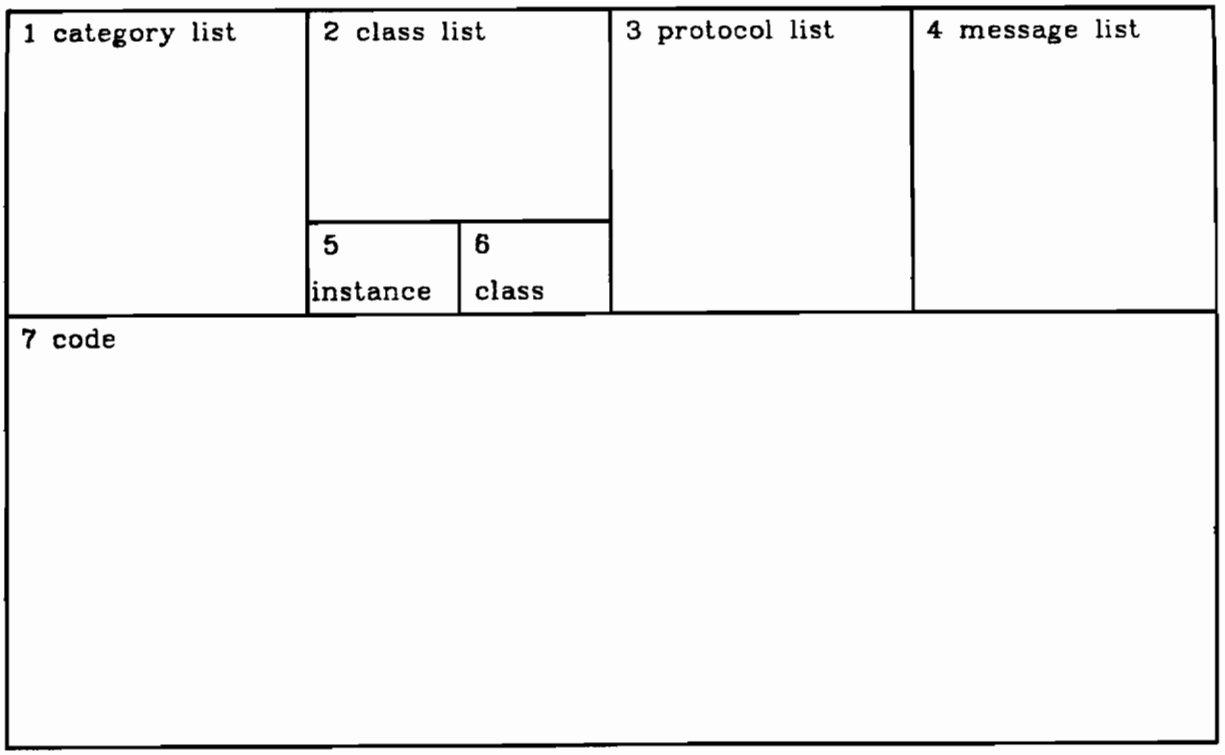


Figure 3.2 System Browser Views

displaying the PrologLiteral method *addBelow*. System browsers have seven views. Figure 3.2 numbers and names each view for purposes of this discussion. View 1 shows a list of categories, which are groups of classes. In Figure 3.1, the category *SIG-Tests* is selected. View 2 shows the classes in the category selected in view 1. Figure 3.1 shows all the classes in the category *SIG-Tests*, with class *PrologLiteral* selected. View 3 shows a list of protocols, which are groups of messages, associated with the class selected in view 2. In Figure 3.1, the protocols for PrologLiterals are shown, with the protocol *textView menu messages* selected. View 4 displays the messages associated with the selected protocol. In figure 3.1, the message *addBelow* is selected from the protocol *textView menu messages*. Views 5 and 6 are called BooleanViews. BooleanViews are either on (black background) or off (white background). Views 5 and 6 complement each other; one is always on, the other off, to indicate whether view 4 shows instance messages or class messages. In Figure 3.1, view 5 is on, so view 4 shows instance messages. When views 1 through 4 all have selections made, view 7 shows the method for the message selected in view 4. In Figure 3.1, the method *addBelow*, that was explained at the beginning of this chapter, is displayed in view 7.

Smalltalk inspectors were introduced in Chapter 1. These IDs are used to inspect and change the instance variables of an arbitrary object. The left view of an inspector displays the list of instance variables owned by the object being inspected plus a pseudo instance variable, *self*. Whenever one of these variables, or *self*, is selected in the left view, the right view displays the value of that variable. Figure 1.1 shows an inspector on a rectangle with upper left corner at the point (100,100) and lower right corner at the point (200,250).[2] Since *self* is selected in the left view of this figure, the right view shows *(100@100 corner: 200@250)*, which depicts the value of the entire rectangle, not just one of its instance variables.

Figure 3.3 shows an inspector on the PrologLiteral *first(11(111, 112), 12)* that we encoun-

[2]Rectangles do not have to be displayed to exist as valid Smalltalk objects.

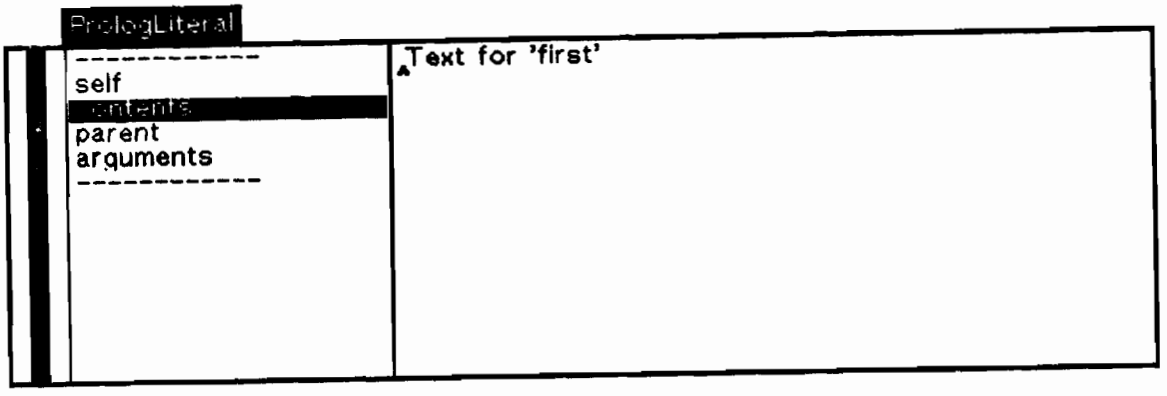


Figure 3.3 Inspection of an Instance of Class PrologLiteral

tered in Figure 2.1. PrologLiterals have three instance variables: *contents*,[3] *parent*, and *arguments*. The right view of Figure 3.3 shows the value of the instance variable *contents* to be *first*, as it should be. Figure 3.4 shows the same inspector with instance variable *parent* selected in the left view. The displayed value of the instance variable *parent* is *a PrologLiteral*. The parent of one PrologLiteral is another PrologLiteral. The text *a PrologLiteral* is the best an inspector can do when asked to depict a PrologLiteral. Figure 3.4 shows the one menu message, *inspect*, available in the left view of an inspector. Selecting menu message *inspect* in Figure 3.4

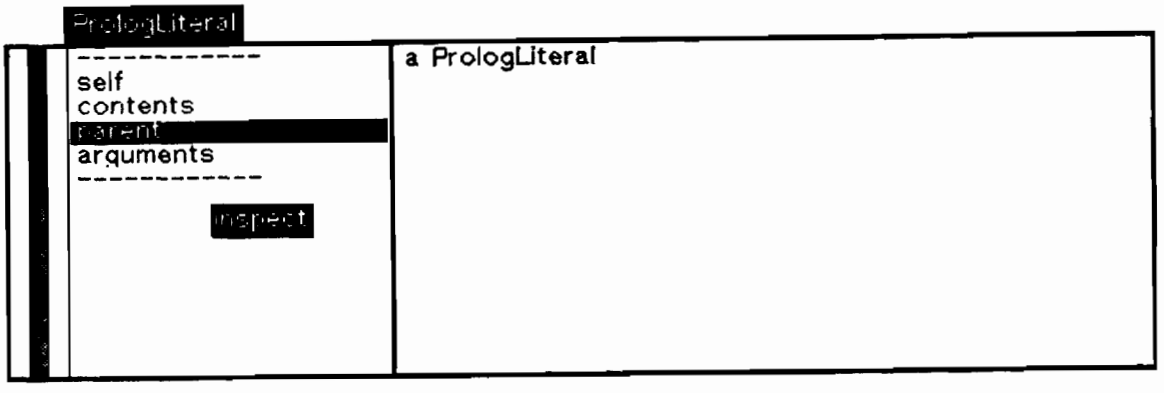


Figure 3.4 Spawning an Inspector

[3]For historical reasons, the instance variable holding the functor of a PrologLiteral is called *contents*.

spawns another inspector on the parent of PrologLiteral *first*. The result is shown in Figure 3.5. In Figure 3.5, we see that the contents (functor) of the parent of PrologLiteral *first* is :-, as we would expect.

Both system browsers and inspectors display data dynamically as objects change. They also allow interactive editing of objects by positioning the cursor on the display using the mouse and entering new data at the selected position via the keyboard. Other kinds of IDs are also supplied with the Smalltalk system, all providing interactive displays with editing.

IDs in Smalltalk are built from three separate kinds of objects. The object that is displayed by an ID is the *model*. The object that displays the data on the screen is the *view*, and the object that takes update commands from the mouse and keyboard is the *controller*. These three objects taken together, I call an *MVC triple*. Figure 3.6 shows the communication paths between the objects in an MVC triple. A view can communicate directly with its controller and model. A controller can also communicate directly with its view and model, but a model can only communicate indirectly with its view.

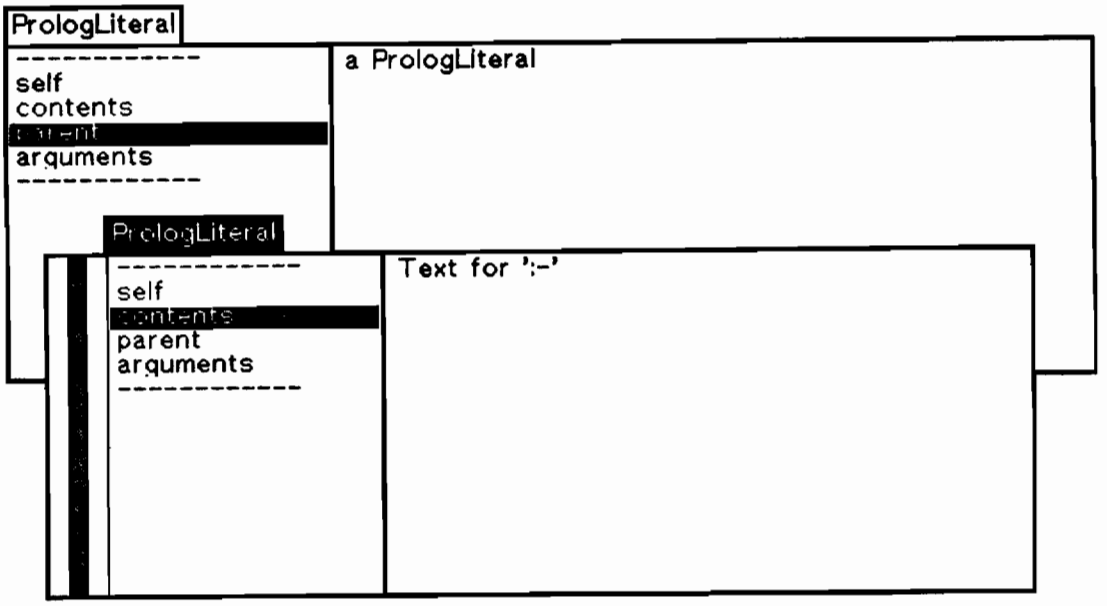


Figure 3.5 A Spawned Inspector

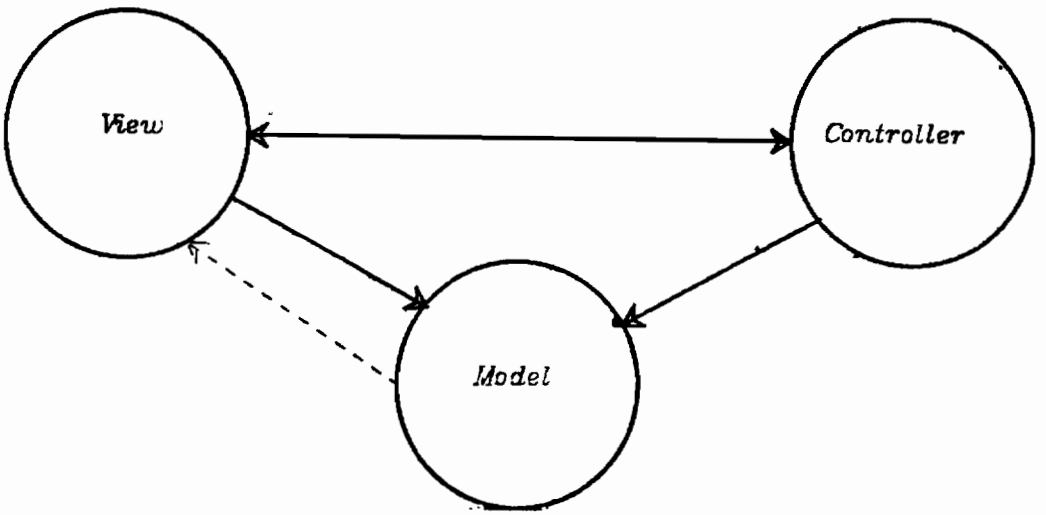


Figure 3.6 Communication Paths in an MVC Triple

### 3.2. Model

The model of an MVC triple can be any object. The code for the MVC paradigm suggests that the model was originally intended to be the logical center of the ID. Since the model can be an instance of any Smalltalk class, there is no view-specific or controller-specific information inherent in it. The functionality of both views and controllers is dependent on the model, but not vice versa.

### 3.3. View

The view displays the model on the screen. Therefore, the view must assume something about the class of the model. Thus, there are many kinds of views, each able to display certain classes of models. There are ListViews, FormViews, TextViews, as well as generic views that can be customized by the user via subclassing. A certain core of operations and data is common to all views, such as their *insideColor*, and the method *displayView* that displays their model. Several methods, including *displayView*, are reimplemented in each new subclass of View, the class that defines the basic instance variables and methods for all views. Two of the

instance variables defined by View contain the view's model and controller.

### 3.4. Controller

The controller controls update and display of the ID. It receives commands from the user via the mouse or keyboard. Controllers for IDs on text monitor both the keyboard and mouse buttons. When the cursor is moved inside such an ID and the red button pressed, a pointer marking the current position in the text is moved. This action is accomplished by the controller. This controller also pops up a menu when the yellow button is pressed, and executes the selected command by passing it on to the model. Figure 2.1 shows such a pop up menu. All controllers have instance variables *view* and *model*.

### 3.5. Composing IDs

The discussion above assumed that one model, one view, and one controller comprised an ID. This simple paradigm is almost never the case; IDs are composed of multiple MVC triples. Complex objects have heterogeneous data and require multiple views and controllers to produce satisfying IDs. Figure 2.1 shows an ID consisting of several MVC triples. Each text item visible in the figure is displayed by one triple, and there are others not immediately visible.

When an ID is composed of several MVC triples, these triples are hierarchically related. All views maintain a list of subviews. Views may have any number of subviews, but only one superview. Thus views in an ID form a tree, called the view hierarchy. The view is the only member of an MVC triple that maintains this hierarchical relationship. Generally, adding an MVC triple to an ID is done by adding a view as a subview of one of the views currently in the ID. All views understand the message *defaultControllerClass*, which returns the class of the normal controller for the view. When a view is assigned a model, this message returns the class of the controller to use.

Having multiple MVC triples introduces problems concerning which controller should have control at any given time. The view hierarchy determines in which order to display views and

which controller should have control. In the following discussion, we differentiate between views as they appear on the screen and views as they are related to the view hierarchy. The words *upper/lower* are used to indicate a view's position on the screen. The words *top/bottom* and *above/below* indicate a view's position or relative position in a hierarchy of views.

Figure 3.7 shows the view hierarchy for the ID shown in Figure 2.1. The numbers indicate the order in which these views were added to the ID. Views that have only a number are invisible in Figure 2.1. Views containing text are visible in Figure 2.1. The dotted lines indicate co-extensive views. Each view in Figure 3.7 has a related controller and model, and thus represents an MVC triple. In the following discussion, MVC triple components are referred to by the number shown in Figure 3.7, e.g., *controller 1* is the controller for the triple at the top of the hierarchy, *view 5* is the view displaying the text :-. When the cursor is initially positioned over the ID shown in Figure 2.1 and any mouse button is clicked, controller 1 gets control. By convention, only the controller at the top of the hierarchy handles the blue button, which allows moving, resizing, and removal of the entire ID. If the blue button is not down, the top controller attempts to pass control down the hierarchy by telling the topmost view to query its subviews in order. Each subview asks its controller if it wants control. Controllers not at the leaves of the hierarchy, usually defer to controllers below them. Controllers at the leaves of the hierarchy generally want control if their view has the cursor. These controllers are responsible for processing red and yellow button commands. Conventionally, the red button is used for selecting items, e.g., an item in a list or a position within a paragraph of text. The yellow button is used to display a menu of view-specific commands, such as *copy*, *cut*, and *paste*, for views editing text.

Using Figure 3.7 as a guide to Figure 2.1, we see that Figure 2.1 shows a scroll bar beside view 21. Scroll bars are displayed by controllers. Scroll bars indicate, among other things that the controller displaying them currently has control.

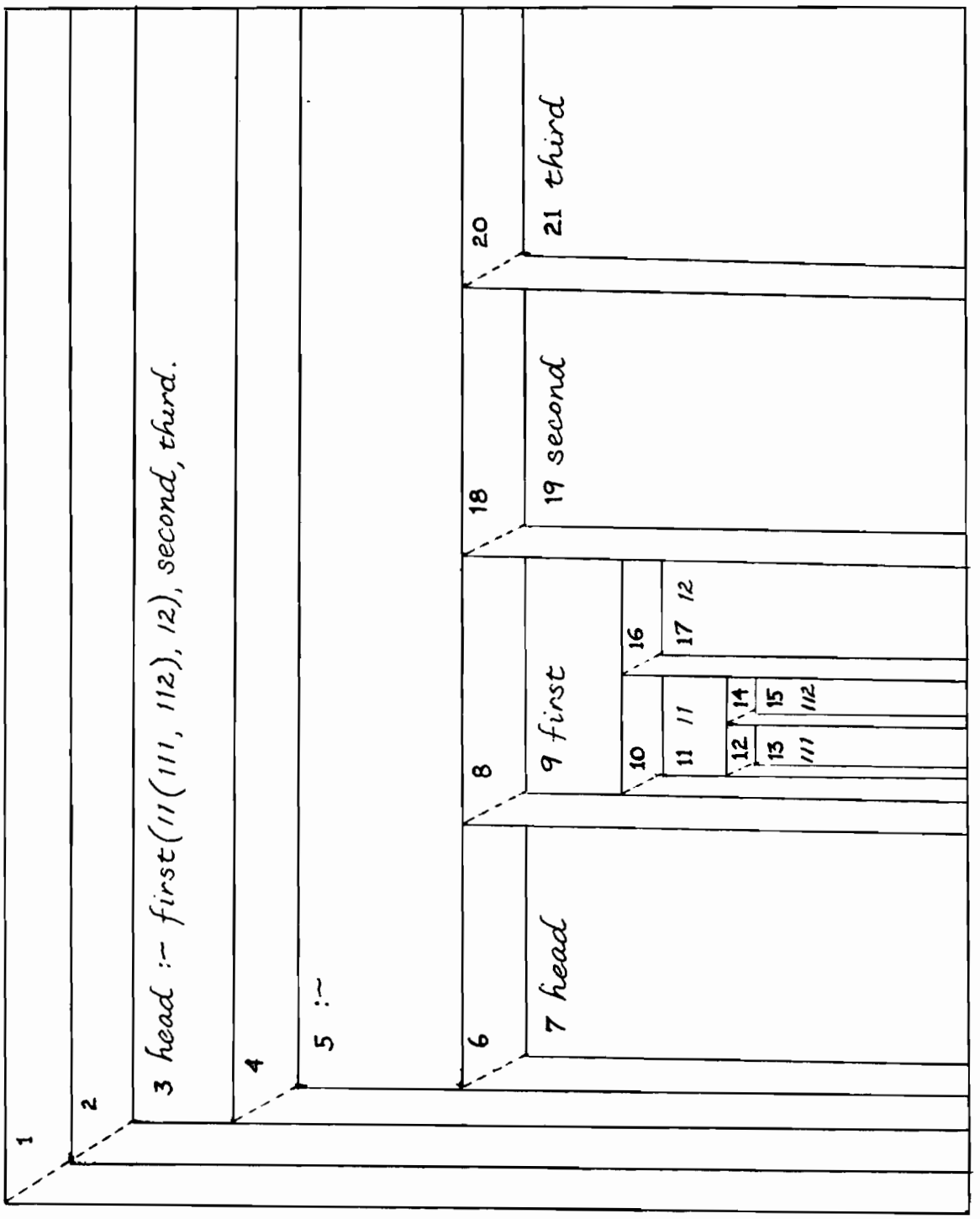


Figure 3.7 Prolog Literal View Diagram

### 3.6. Dependency Table

Models directly reference neither their view nor their controller. However, on occasion, models do need to tell views to redisplay. Suppose we want to display the same object in two different places in an ID and allow editing either place. Two MVC triples in the ID would have the same model, but separate views and controllers. If controller A makes a change to the model view B is now displaying old information. The only thing logically linking views A and B is their model. The model needs to notify all views on it of changes. Smalltalk maintains a table relating views to models. Each view in the system is recorded in this table. When an object that is the model of some view issues the message *self changed* each view having that object as model receives the message *update:*. On receipt of the message *update:* each view takes appropriate action, usually redisplaying itself.

### 3.7. Problems

The MVC paradigm described above does work, but some aspects are more awkward than they might be. When building an ID under this paradigm, a major problem is where to put what. Operations can be performed in the model, view, or controller. Deciding when the controller should take action itself and when it should defer to the view or model can be difficult. Where data should be stored is another question. The view can, and often does, store model-specific data, but to do so, it and the model must be highly integrated. What usually happens to effect this integration is the creation of a view subclass and a controller subclass for every new kind of ID. For example, the first implementation of the ID for Prolog literals shown in Figure 2.1 used subclasses of StringHolderController and StringHolderView, which are classes for editing text objects. In this first implementation, PrologLiteral was a subclass of StringHolder, the class of models for StringHolderViews and StringHolderControllers. StringHolderControllers implement text editing menu messages. However, controllers in a Prolog literal ID must also deal with the menu messages for manipulating a tree of PrologLiterals, such as *add below*, *add left*, and *add right*. StringHolderViews were also insufficient because they can not be

selectively updated. In the Prolog literal ID, the PrologLiteral at the top of the tree of Prolog literals, the one whose functor is :-, is the model for both view 3 and view 5 in Figure 3.7. In order to update view 3 when other views change, in the current implementation, the PrologLiteral at the root of the tree executes *self changed: #clauseAsText.* This action causes both views 3 and 5 to receive the message *update: #clauseAsText.* On receipt of this message, only view 3 redisplays. In the first implementation, a subclass of StringHolderView was needed to recognize a special parameter with the *update:* message. The second implementation does not use StringHolder, StringHolderController, or StringHolderView. Instead, a version of pluggable views are used. Pluggable views allow selective update without having to create special view subclasses.

### 3.8. Pluggable Views

Pluggable views are provided with Smalltalk-80 version 2 as an aid for creating user interfaces. Models for pluggable views encompass more information. Generally, a pluggable view displays only a part of its model. In Figure 3.3, we saw an ID that is a standard inspector with a view listing instance variables and a view showing the value of the selected instance variable. These views are pluggable and have the same model, an Inspector. Pluggable views also make selective update easier. Again considering an inspector ID, when a new instance variable is selected, the model executes *self changed: #text* which updates the text view, but leaves the list view untouched. Pluggable views reduce subclassing of views and controllers. Menu messages for the yellow button are sent to the model. Instead of a specialized controller to implement messages peculiar to the model, a standard controller passes these messages to the model. In Figure 2.1, the messages *add below, add left, add right, delete, format,* and *expand* are tailored to PrologLiterals, and are executed by instances of class PrologLiteral. In the first implementation, PrologLiteralController, a subclass of StringHolderController, was created solely to pass the messages above on to the model, a PrologLiteral.

The features described make pluggable views attractive for developing code. Fewer classes are created; fewer decisions must be made concerning where to implement ID behavior; and debugging is simplified.

### 3.8.1. System Supplied Pluggable Views

Currently, the system supplies these views in three flavors: Boolean, text and list, called BooleanView, TextView, and SelectionInListView, respectively. The related controllers are SwitchController, TextController, and SelectionInListController. BooleanViews act as indicators that a certain condition is true or false. TextViews hold text and TextControllers perform the text editing functions; the subclass CodeView is specialized to hold text that can be interpreted as Smalltalk code. TextControllers store a copy of the displayed text during editing. This text is only sent to the model when the user issues the *accept* command. SelectionInListViews hold a list of items and allow one item at a time to be selected from the list.

### 3.8.2. Usage

Models communicate with pluggable views via the *changed:* message, and pluggable views pass messages to models, corresponding to the various actions taken during an ID. I call these messages *creation messages* because the view is told, when it is created, the actual message *selectors* to use when passing messages to the model. Each flavor of pluggable view has a different set of creation messages, but all three have *aspect* and *change* messages. Note that *aspect* and *change* are not the message selectors; they are terms for a kind of creation message. I will discuss these messages first as they relate to all three views, then discuss the other messages needed by each view in turn.

#### 3.8.2.1. Aspect Message

The aspect message selector serves two vital roles for pluggable views. The model uses the selector of the aspect message to signal the pluggable view to redisplay. The pluggable view

sends the aspect message to the model to retrieve information for that display. Thus, the aspect message indicates what aspect, or part, of the model has changed. In a typical ID, a change occurs in the model; the model then executes *self changed:* <*aspect selector*> where <*aspect selector*> is the aspect message associated with some pluggable view on this model. A model uses *self changed:* <*aspect selector*> when a change in one view necessitates changing other views. For example, the model for an inspector ID is an instance of class Inspector. In an inspector ID, when the list view selection is changed, the text view in the ID must change. The model accomplishes this update by executing *self changed: #text*. (Here *#text* is the aspect message for the text view.) Both the list view and text view thereby receive the message *update: #text*, but only the text view was created with aspect message *text*, so only the text view redisplays.

When a pluggable view receives the message *update:* <*selector*>, it checks to see if <*selector*> is its aspect message. If it is, the view promptly sends <*selector*> as a message to its model to retrieve the data needed for display. The model's method for this aspect message returns data that is used by the pluggable view for redisplaying. To continue the example at the end of the last paragraph, when the text view of the inspector ID receives the message *update: #text*, this view first checks that *text* is its aspect message. The view then sends the message *text* to the Inspector that is the model for the ID. In response to *text*, the Inspector returns a Text[4] object that represents the value of the instance variable selected in the list view of the ID.

Each of the three pluggable views expects its model to return something different in response to the aspect message. TextViews expect to receive a Text. BooleanViews expect true or false. SelectionInListViews do not use the aspect message to retrieve data from the model; instead, they have a separate message for this purpose.

---

[4]Text is a Smalltalk class whose instances are text strings. Texts have font information whereas Strings are only an array of Characters.

#### 3.8.2.2. Change Message

A pluggable view sends its change message to its model as a result of a specific action taken by the user. For BooleanViews, this action is moving the cursor over the view and clicking the red button. For TextViews, it is executing the menu item *accept*. For SelectionInListViews, it is selecting a new item in the list. The change message is always sent with at least one argument, which indicates the exact update the model should perform. For BooleanViews, the argument is a Boolean value. For TextViews, the argument is the text currently displayed in the TextView; and for SelectionInListViews, it is the item currently selected in the list. Again consider an inspector ID; the change message for the list view is *field:*. When the user selects a new instance variable *var* from the list, the list view sends the message *field: var* to its Inspector. Inspectors themselves have an instance variable, *field*, which is reset to *var* on receipt of this message.

#### 3.8.2.3. TextView

TextViews have no creation messages other than the aspect message and the change message. In a TextView, when the user selects menu item *accept*, the view sends to its model the change message with the new text as an argument. The change message for a TextView may take two arguments. When the change message has only one argument, the TextView sends its model only the text currently displayed in the view. When the change message has two arguments, the text and the TextController associated with the TextView are both sent to the model. Most models will want to receive only the text.

Concerning the text object that is being edited by a TextView-TextController pair, TextControllers continue to work on the actual text object sent to the model with the change message, so a word of caution is in order. If the model needs to store the text received with the change message, this model must store a copy of the received text. If this copy is not made, the user will be editing the actual text stored in the model. In general, the user wants to edit a

work copy of the text. Similarly, when a model receives a request for text to be displayed, it should respond with a copy of the text.

TextViews also respond to the message *updateRequest* which, is sent to all of a model's dependent views when the model sends itself the message *changeRequest*. On receipt of *updateRequest*, a TextView first checks whether the displayed text has been edited since the last *accept*. If so, a confirmer appears with the message

> The text showing has been altered.
> Do you wish to discard those changes?

to which the user must answer yes or no. Figures 3.8 and 3.9 show an example of this scenario. Figure 3.8 resulted from removing all the text showing in the CodeView of Figure 3.1 then selecting the message *addLeft* in the message list view. The confirmer appeared because the method for message *addBelow* has been changed but not accepted. Figure 3.9 shows the result of answering *yes* to the confirmer shown in Figure 3.8. The original method for *addBelow* was left intact, and the method for *addLeft*, the new selection, is shown in the CodeView.

#### 3.8.2.4. BooleanView

BooleanViews require two creation messages in addition to the aspect message and the change message. In reality these are not messages that will be sent by the view to the model, but parameters to be used by the view. The first is the *label parameter*, which is the string that is displayed inside the BooleanView. The two BooleanViews in system browsers have labels *instance* and *class*. The second is the *value parameter* which must be true or false. This parameter is sent to the model as the argument with the change message, to indicate how the model should change state.

#### 3.8.2.5. SelectionInListView

In addition to aspect and change messages, SelectionInListViews must have a *list message*, and may have a *print-items parameter*, *one-item parameter*, and an *initial-selection message*.

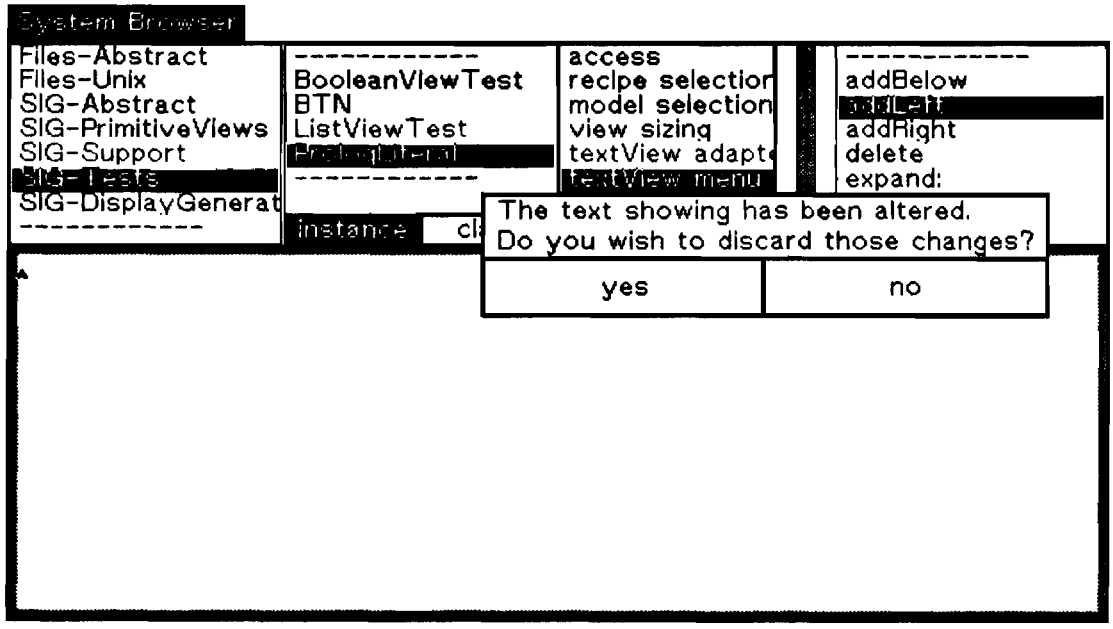


Figure 3.8 A Confirmer

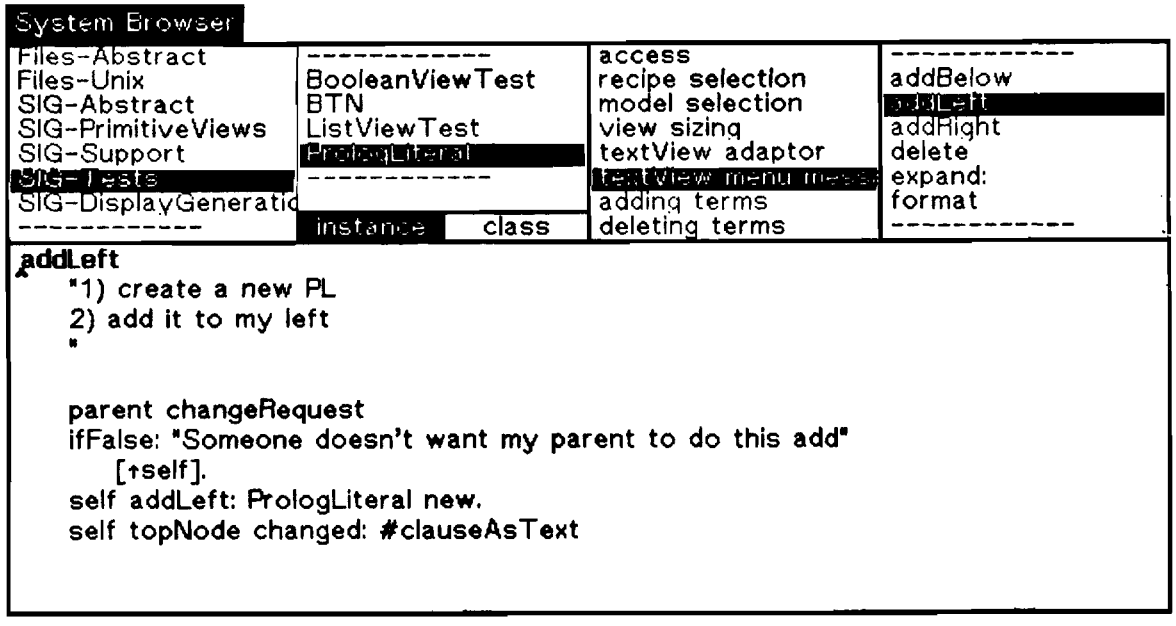


Figure 3.9 Answering *yes* to the Confirmer

As mentioned above, the aspect message for SelectionInListViews is not expected to return information. Instead, the view sends the list message to the model to retrieve the list of objects

to show in the view. The print-items and one-item parameters are Boolean values that signal the SelectionInListView to act in certain ways. The one-item parameter is true only if there will always be one item in the list and that item is always selected. The print-items parameter indicates if the result of sending the list message to the model is a list of strings, which can be displayed directly, or is a list of objects, for which string representations must be obtained. A view sends the initial-selection message to its model whenever the view is redisplayed. When the model receives the initial-selection message, it returns one of the elements in the list to be displayed. The SelectionInListView then selects the returned item in its display.

#### 3.8.2.6. Menu Message

TextViews require, and SelectionInListViews allow, one other creation message, the *menu message*, which specifies the menu that appears when the yellow button is depressed. When the model receives the menu message, it must return an ActionMenu.[5] For SelectionInListViews, any messages specified in this menu are passed to the model, so the methods for these messages should be implemented by the model. TextViews require that the menu message return a menu containing a full compliment of editing functions (*again*, *undo*, *copy*, *cut*, *paste*, *do it*, *print it*, *accept*, *cancel*) if you want them to appear in the TextView's menu. However, these messages are implemented in TextController or CodeController, so the model never receives any of these messages. If additional messages are added to this menu, they are passed to the model, so the model must implement methods for them.

[5]ActionMenu is a subclass of PopUpMenu; the new functionality it gives makes it easier to associate menu labels with the message selectors that implement those labels.

# 4. Smalltalk Interactive Display Generator

I wrote the Smalltalk Interactive Display Generator (SIG) to create a logical extension of pluggable views. Pluggable views are used extensively in the standard Smalltalk system in browsers and inspectors, to standardize and shorten the code used for the Smalltalk interface. Browsers, Inspectors, Debuggers, ChangeLists, FileLists, Workspaces are all hard-coded IDs. That is, each of them is coded in a different way, specific to the application they are performing and the programmer who wrote them. In particular, they all implement different kinds of top views, each a subclass of StandardSystemView.

The standard IDs mentioned above use a fixed set of subviews. The number or type of subviews does not change in response to changes in the data structure being viewed. Inspector IDs always have two visible views, file lists three, and system browsers seven. Though pluggable views do provide a standard way to separate models from view-controller pairs, in some ways pluggable views limit the kind of IDs that can be produced. Pluggable views are designed to occur at the leaves of a view hierarchy, and thus are not meant to have subviews themselves. Almost all the system IDs have a two level hierarchy: a top view with subviews. While this arrangement works fine for many applications, one needs to be able to add subviews dynamically depending on the state of the model (to have a general mechanism for creating IDs). SIG provides these two features: a standard mechanism for adding subviews, and variations in the number, size, and type of subviews based on the current state of the model. The following is a discussion of SIG's components and how they interact to provide these features.

## 4.1. Separate Place for ID Specifications

For the first feature, I postulated the need for a "language" for specifying the views that appear and the yellow button menus available within those views. Recall that a Smalltalk ID

consists of not only yellow button menus, but also red button and keyboard activity. However, defining yellow button menus is sufficient for many IDs and SIG allows for the definition of red button and keyboard activity also.[1] This language is interpreted by SIG to create and manage IDs on data structures. Writing a "program" in this language consists of creating a set of display types. Executing this program is accomplished by telling SIG to open an ID on an object using one of the display types in the set.

A display type is simply a data structure that holds the parameters necessary for specifying an ID; it is similar in this respect to the *interesting events* described by Brown [Brow84]. display types are separate from the MVC triples that make up the ID itself. This arrangement is different from the pluggable view paradigm wherein all parameters for the ID are kept in the view itself. The display type is a separate, standardized entity that provides these parameters to the pluggable views. Heretofore the model and special views stored these parameters.

### 4.2. Structure of an ID

An ID generated by SIG consists of multiple MVC triples. The ID is opened with a single MVC triple. Then, other MVC triples are added to that triple using a display type for the model associated with the first triple. For all IDs, one or more display types specify the classes of the models, views, and controllers that make up the MVC triples in these IDs. The view component of each MVC triple is an instance of a view class provided with the SIG system.[2] All the view classes provided by SIG are pluggable in the sense that they are parameterized. SIG views expect their models to understand certain kinds of messages. The possible kinds of messages a view may send to its model include: a *recipe message*, a *model message*, a *rectangle message*, a *creation message*, a *menu item message*,[3] and a *display message*. A display type specifies the actual selectors for these kinds of messages to be sent to a model. Figure 4.1 shows

[1] See User Defined Primitives in this chapter.

[2] Users may also define new view classes that can be incorporated into SIG.

[3] Controllers, not views, send menu item messages to the model.

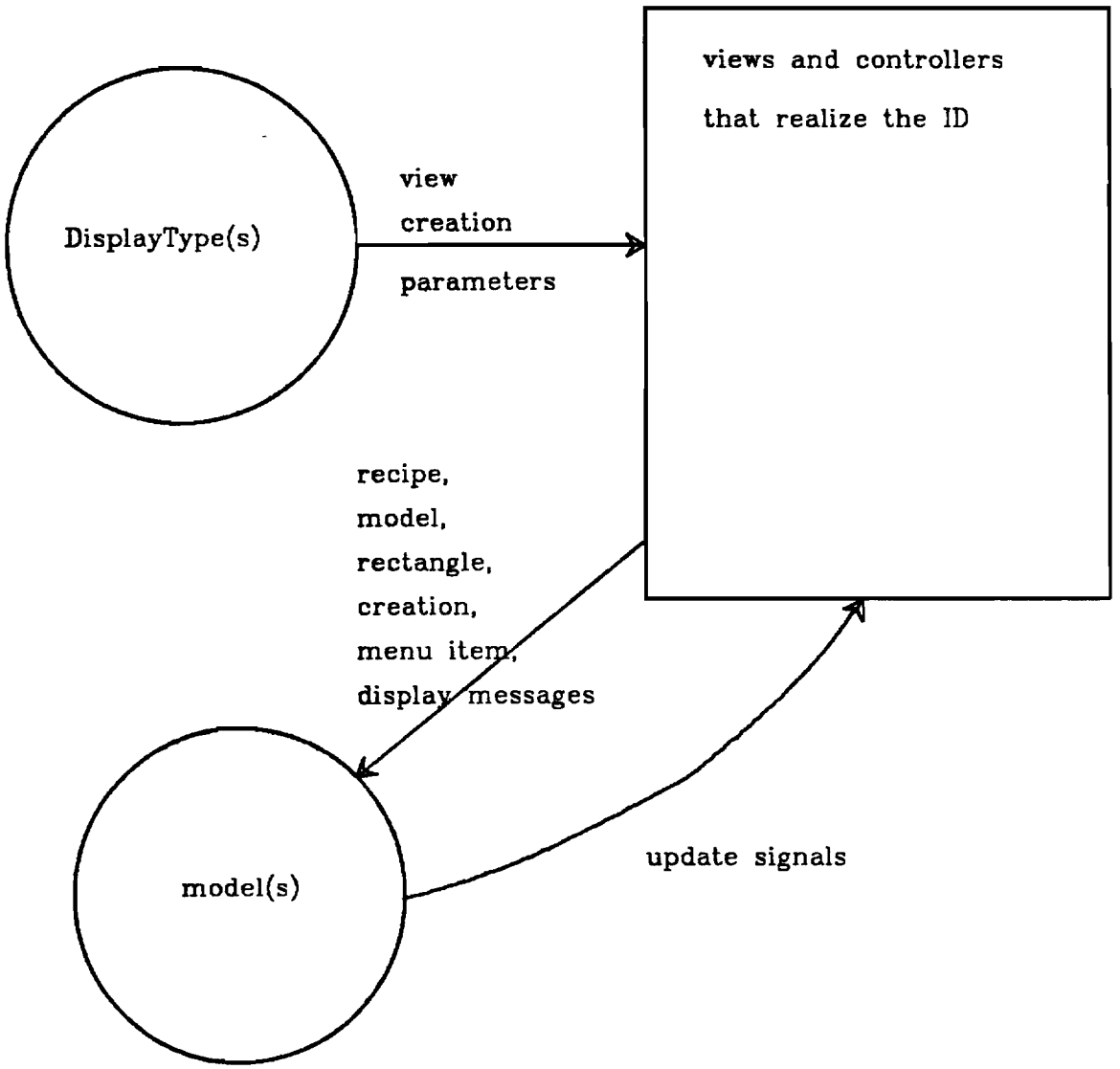


Figure 4.1 Components of an ID

that the views and controllers of MVC triples in an ID extract message information from the display types describing an ID. This information is used to send messages to the models of these MVC triples. In addition, models signal their views to update as described in Chapter 3. Display types and models are supplied by the user. Display types reflect the structure of their models, and the models must understand the messages specified in their display types. Figure 4.2 shows an ID on a binary tree, an instance of class BTN. The class is called BTN because

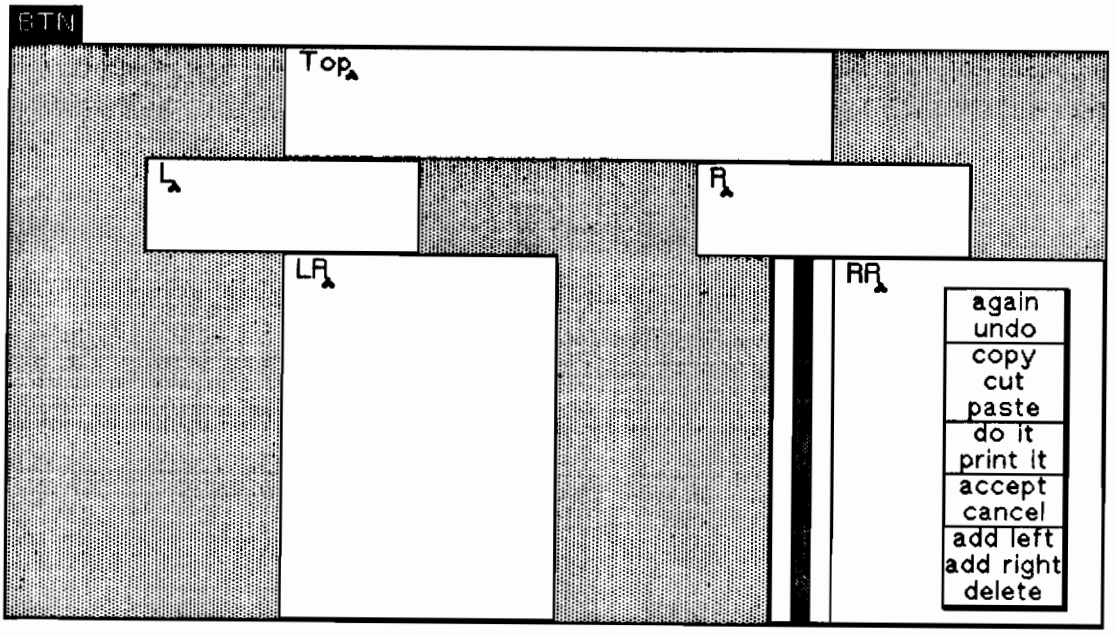


Figure 4.2 ID Using Display Type *withoutArrows*

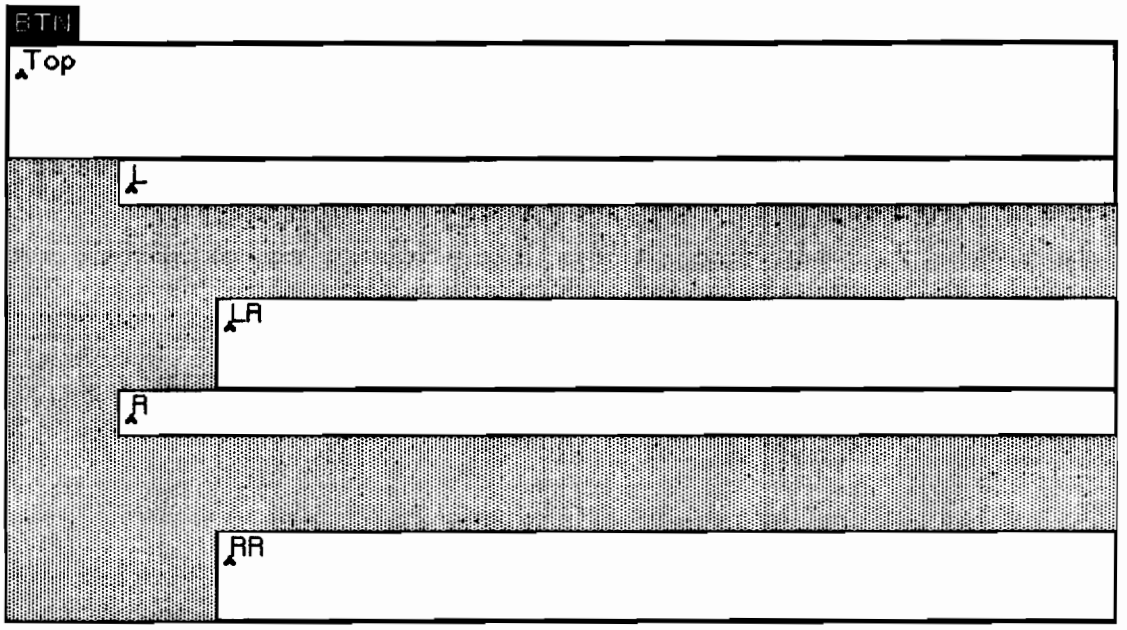


Figure 4.3 An ID Using Display Type *outline*

each instance is a node in the tree. A node has data, a block of text, a parent, a left child, and a right child. The parent, left child, and right child are themselves instances of class BTN. By

following these references, we can go from any node in a binary tree to any other node in the tree. The display type used in this ID is called *withoutArrows*. Figure 4.3 shows an ID on the same binary tree. The ID shown in Figure 4.3 is specified by a display type called *outline*. There are five visible views in Figure 4.2. Each has a binary tree node as its model. The root node is the one on which the ID was opened. The system added views of the other nodes after looking at the root node. The display type *withoutArrows* contains four *recipes*, each of which specifies a different way to add subviews to the ID. A particular recipe is selected on the basis of the number of non-nil children of the node. If a node is a leaf, *withoutArrows* adds one MVC triple, which can display and manipulate the node's label, to the current ID. The model for this MVC triple is the leaf node itself. In Figure 4.2, the subviews showing the texts *LL* and *LR* are views on leaf nodes. If a node has a right child but no left, two MVC triples are added to the ID: one to hold the node's label and one to hold the right subtree. This case holds for nodes *L* and *R* in Figure 4.2. If a node has a left child but no right, two MVC triples are added to the ID: one to hold the node's label and one to hold the left subtree. Finally, for a node with 2 subtrees, three MVC triples are added: one to display the node's label, and one each for the left and right subtrees. Node *Top*, the root node of the binary tree, uses this recipe. The display type also contains the menu available to any view displaying the label of a node. This menu is shown in Figure 4.2 for the view displaying the node *RR*.

Display types store the recipes mentioned in the example above. The display type has a condition associated with each recipe. The condition takes the form of a message sent to the model to determine its state. For an instance of class BTN, we send the message *isLeaf*, which returns true if the BTN represents a leaf node and false otherwise. Each recipe consists of one or more specifications for subviews to add.

### 4.3. Structure of a Display Type

DisplayType is a class defined in the SIG system to represent display types. The following sample DisplayType is called *BLTest1*. It specifies IDs on instances of the class BooleanListTest, which is an innocuous class created solely for the purpose of illustrating display types. Figures 4.4a and 4.4b show two pictures of an ID specified by the display type *BLTest1*. The ID has either 2 Boolean views or two Boolean views and a list view, as shown in the figures. This ID simply makes the list view appear and disappear when the appropriate Boolean view is highlighted. Each display type in SIG is associated with one Smalltalk class, but one class may

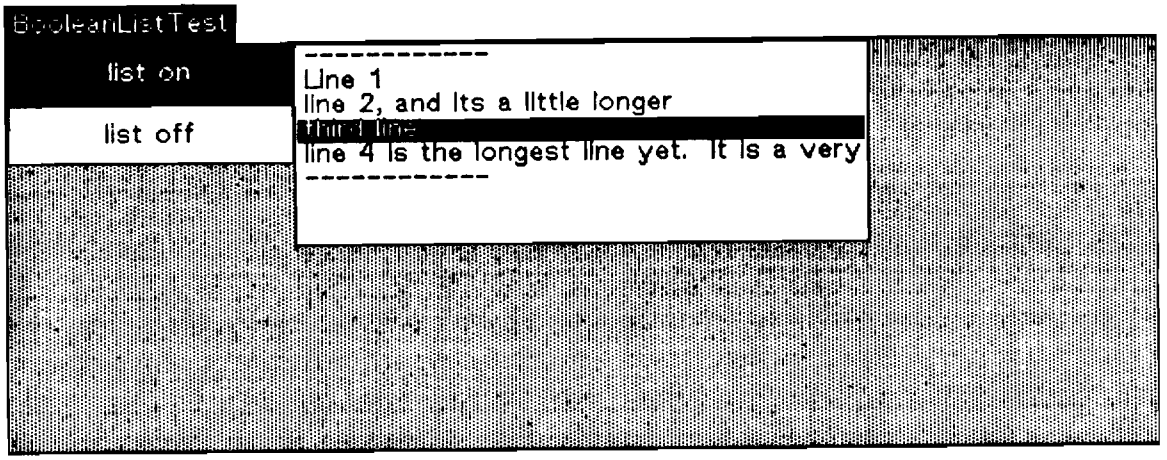


Figure 4.4a BooleanListTest With List On

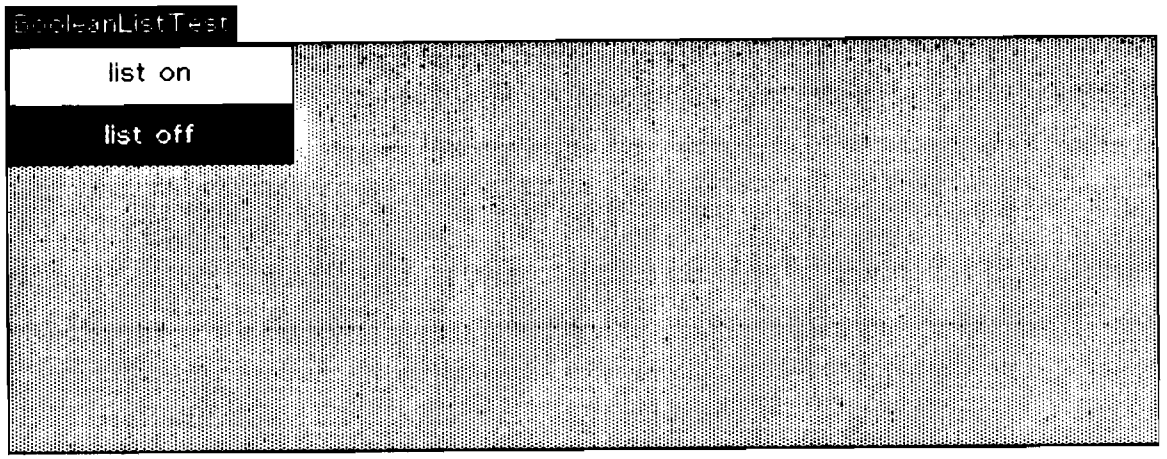


Figure 4.4b BooleanListTest With List Off

have many display types. Display types are composed of recipes, and recipes are composed of *specifications*. The classes MVCRecipe and MVCSpecification, that represent recipes and specifications, are explained below. The instance of DisplayType describing *BLTest1* is listed in Figure 4.5 in outline form. The class name, BooleanListTest, appears in bold print at the left margin. The name of the display type, *BLTest1* appears under the class BooleanListTest. Two recipes appear under the display type. The recipes are headed by their associated conditions *isListOn→()* and *notNil→()*. Under recipes are lists of specifications for subviews denoted by the recipe. For each specification, the class of the subview appears first, with the rest of the specification indented below it.

Each instance of class DisplayType has the instance variables *class*, *name*, *default*, and *recipes*. The instance variable *class* contains the class displayed using this DisplayType. Instances of class BooleanListTest can be displayed using DisplayType *BLTest1*.[4] For the DisplayType in the binary tree example above, the instance variable *class* contains the object *BTN class*.[5] The instance variable *name* contains the name of the DisplayType. DisplayType *BLTest1* has *name* 'BLTest1', obviously display types on a class must have unique names. The instance variable *default* indicates whether a DisplayType is the default for its class. The instance variable *default* is true for *BLTest1*. The final instance variable *recipes* is a collection of MVCRecipes. Each MVCRecipe consists of a Boolean message, for its condition, and a list of specifications for MVC triples. If the condition is true for a model, one MVC triple is created for each specification in the list, and added to the current ID. The collection of recipes is ordered. The recipe selected from the collection is the first one whose condition is true of the model. If all conditions are false, an error occurs. In the DisplayType *BLTest1*, the instance variable *recipes* is a collection with two MVCRecipes, each containing a different list of MVC triple specifications.

---

[4]Currently, SIG does not allow instances of subclasses of BooleanListTest to be displayed using DisplayType *BLTest1*.

[5]Class describing objects are often denoted by the class name followed by the word *class* [Gold83], [Gold84].

**BooleanListTest**

*BLTest1 - default*

*#isListOn→()*

```
MVCBooleanView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0@0 corner: 0.25@0.15
    creationMessages: Dictionary (#label→'list on' #aspect→#value
        #change→#value: #value→true )
    menu: nil
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
    displayMessage: nil
    controllerClass: nil

MVCBooleanView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0@0.15 corner: 0.25@0.3
    creationMessages: Dictionary (#label→'list off' #aspect→#value
        #change→#value: #value→false )
    menu: nil
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
    displayMessage: nil
    controllerClass: nil

MVCListView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0.25@0 corner: 0.75@0.5
    creationMessages: Dictionary (#printItems→false #change→#selection:
        #initialSelection→#selection #aspect→#list #oneItem→false )
    menu: nil
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
```

Figure 4.5 Display Type *BLTest1*

```
        displayMessage: nil
        controllerClass: nil

#notNil→()

    MVCBooleanView
        modelMessage: #yourself
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0@0 corner: 0.25@0.15
        creationMessages: Dictionary (#label→'list on' #aspect→#value
            #change→#value: #value→true )
        menu: nil
        subject: nil
        leftBorder: 1
        rightBorder: 1
        topBorder: 1
        bottomBorder: 1
        displayMessage: nil
        controllerClass: nil

    MVCBooleanView
        modelMessage: #yourself
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0@0.15 corner: 0.25@0.3
        creationMessages: Dictionary (#label→'list off' #aspect→#value
            #change→#value: #value→false )
        menu: nil
        subject: nil
        leftBorder: 1
        rightBorder: 1
        topBorder: 1
        bottomBorder: 1
        displayMessage: nil
        controllerClass: nil
```

Figure 4.5 Display Type *BLTest1* continued

### 4.4. Structure of an MVCRecipe

MVCRecipe is a subclass of Association, which is a class that associates two objects. MVCRecipes associate a condition with a list of MVCSpecifications. Associations have a *key* and a *value*. The key for an MVCRecipe is the condition, which is represented as a message with a list of arguments, called the *condition message*. Condition messages are sent to a model to determine which recipe to use. If the model receiving the message returns true, the recipe

*matches*. Condition messages in MVCRecipes are represented as

message selector→(arguments)

A recipe is usually denoted by its condition message. Thus, the recipes in *BLTest1* are denoted, *isListOn→()* and *notNil→()*. As we can see, both condition messages have empty argument lists. In Figure 4.4a, the recipe *isListOn→()* matches. In Figure 4.4b, the recipe *notNil→()* matches. The message *notNil* is often the condition message for the last recipe in a collection of recipes because it will always return true.[6]

The value part of an MVCRecipe is a collection of MVCSpecifications. Each MVCSpecification represents one MVC triple. Recipe *isListOn→()* in *BLTest1* has three MVCSpecifications in its collection. These MVCSpecifications specify the two Boolean views and the list view, and their related controllers. Likewise, the recipe *notNil→()* has two MVCSpecifications: one for each Boolean view. An MVCSpecification always specifies exactly one MVC triple. We next discuss the kinds of views and controllers that SIG supplies for use in MVCSpecifications. All SIG supplied view classes begin with 'MVC' to separate them from standard Smalltalk classes.

### 4.5. Top Views

Top views are instances of MVCTopView, which is a subclass of StandardSystemView. All SIG IDs have a top view, which is at the top of the view hierarchy for the ID. Top views have tabs that display white on black when the ID is active. Top views are not pluggable, because pluggable views can only be subviews, and top views are not a subview of any view. Controllers for top views, instances of MVCTopController have the standard blue button menu for moving and sizing the ID.

All SIG supplied views contain the MVCSpecification from which the view was built. Top views are built the same way for all IDs. Their model is the object on which the ID was opened

[6]Condition messages are not sent to nil objects.

and the rectangle they occupy is specified by the user. A top view has no yellow button menu, has a fixed border width (1), and has an instance of MVCTopController as a controller. Since all these items are fixed, the MVCSpecification in a top view does not describe the top view itself; instead it describes the top view's lone subview. Both a top view and its lone subview contain the same MVCSpecification.

The single subview of a top view is an abstract view, an instance of MVCAbstractView. This subview is called the *seed* abstract view because it adds all the rest of the MVC triples needed for the ID. Generating this subview is not difficult, again, because so many parameters are fixed. Since top views have only one subview, the rectangle the subview occupies is the same as for the top view. Since this subview adds all other MVC triples in the ID, its model is the object on which the ID was opened, which is also the top view's model. To add the rest of the MVC triples required by the ID, the only other information the seed abstract view needs is the display type that specifies how these MVC triples are to be added. The MVCSpecification stored by both the top view and the seed abstract view contains the display type's name. Neither the top view nor the seed abstract view reflect the overall structure of the model for the ID, they simply provide a framework on which to build an ID that does reflect this structure.

### 4.6. Abstract Views

Abstract views, instances of MVCAbstractView, are the major workhorse and the major innovation of the SIG system. They allow IDs to be composed of sub-IDs, IDs to be generated from specifications, and MVC triples to be added conditionally. Smalltalk supplied IDs all have a fixed set of MVC triples. Conditionally adding views while an ID is in progress can not be done using Smalltalk supplied pluggable views, without writing a special top view. The disappearance and reappearance of the list view in the BooleanListTest ID shown in Figures 4.4a and 4.4b demonstrates conditional view adding in SIG. The binary tree and Prolog literal IDs show how IDs are composed by adding MVC triples as needed to reflect the current state of the model. SIG IDs require the user to write less code because the way MVC triples are composed

in an ID is specified (using display types) rather than hard coded. Finally, SIG IDs allow multiple-level view hierarchies, as evidenced by the binary tree and Prolog literal IDs. While multiple-level view hierarchies are not forbidden, it is rare to find a Smalltalk supplied view hierarchy with more than two levels.

Abstract views act as holders for MVC triples that actually carry out an ID. In the BooleanListTest example, there is one abstract view (the seed abstract view) that serves to hold the entire ID. In the binary tree example, the views for each node are subviews of an abstract view; each full node has three views, one text view for the label, and two abstract views to propagate sub-IDs on the left and right subtrees.

### 4.6.1. Composition of IDs

For all IDs, the view just below the top view in the view hierarchy is an abstract view. This view's model is the model for the entire ID. The specification that describes this abstract view is called the *seed specification*; the only item of interest in this specification is a display type name. The named display type together with the model, gives the abstract view enough information to build the ID. IDs for complex objects can be specified using multiple display types connected by abstract views. The Prolog literal ID in Figure 2.1 uses two display types; the first, *PrologClauseBrowser*, has one recipe with two specifications. The first specification describes the uppermost view on the page, a read-only text view; the second specification describes an abstract view that propagates a sub-ID on the Prolog literal itself. The display type *PLOriginal* specifies the recursive part of the ID, i.e., the MVC triples displaying the functor and arguments of each Prolog literal. To specify this recursive display, the display type *PLOriginal* uses several abstract views that each use a display type to add subviews. The display type for each subview is again *PLOriginal*, since we want to display each Prolog literal the same way. In recursive IDs, such as those for binary tree or Prolog literal, each abstract view starts a sub-ID, just as the seed abstract view is the starting point for the interaction at the top level.

The controllers for abstract views normally do not have yellow button menu messages. An abstract view expects that the controllers for its subviews receive the yellow button messages needed in an ID. However, if, as in the binary tree example, part of an abstract view is visible on the screen, unobscured by any of its subviews, yellow button menu messages can appear in that space. The controller for an abstract view, which is an instance of MVCAbstractController, handles menu messages by first trying to pass control down the view hierarchy. If the cursor is positioned inside an abstract view and no subview is under the cursor or the subview there refuses control, the abstract view's controller will pop up a yellow button menu if a menu is specified for the abstract view.

### 4.6.2. Matching Models to Display Types

Abstract views are responsible for combining the objects and the display types describing the ID, to produce the MVC triples actually present on the screen. The method *addSubViews* for class MVCAbstractView accomplishes this combination by deleting and adding MVC triples. When the ID shown in Figure 4.2 starts, the top abstract view has no subviews. It first displays itself. Since the inside color of abstract views is light gray, this action fills the ID with a light gray background. The abstract view then sends in turn the condition message for each recipe in the display type *withoutArrows* to its model, the node with label *Top*. In this case, the first three condition messages *isLeaf*, *isLeftChildNil*, and *isRightChildNil*, return false. However, the condition message for the fourth recipe, *notNil*, returns true. Therefore, the recipe *notNil→()* matches. This recipe contains a list of three specifications, each describing one MVC triple. The first MVC triple displays the label for node *Top*. To add this MVC triple to the ID, the abstract view:

(1) determines the relative rectangle (0@0.25 corner: 0.75@0.2) for the view component, a text view,

(2) determines a model for this text view by sending the model message *yourself* [7] to the abstract view's own model,

(3) creates a new text view,

(4) adds this view as a subview of itself,

(5) tells the text view to initialize itself with the specified creation messages and border,

(6) and displays the new text subview. The abstract view for the *Top* node then adds the second subview (actually, MVC triple) specified in the recipe. This subview is itself an abstract view, but no matter, the procedure is the same. First determine the rectangle the new view will occupy; then determine the model, this time by sending the model message, *leftChild*, to the current model (node *Top*); then create a new abstract view; add it as a subview; and instruct it to initialize itself. The third subview for node *Top* is also an abstract view. This subview is added to the ID in the same way the second one was. The only differences are that the third subview has a different relative rectangle and a new model message, *rightChild*.

To initialize any MVC view, this view must store the specification that describes it. The MVC view then reads this specification to determine its border and creation messages. On initialization, abstract views also execute *self addSubViews*, to add their subviews.

### 4.6.3. Updating Views

Typically during an ID, the user selects a yellow button menu message, e.g., *add left* in BTN ID, which is then sent to the model, a binary tree node. The model then carries out the action required by the message, adding a new node as the left child of itself. Now we are faced with the problem of updating the views on the node. In the BTN example, the node that added the new child sends the message *self changed* to request the update. The methods implemented by class BTN are shown in appendix A. The method *addLeft* under protocol *tree functions*

[7] The message *yourself* returns the object to which it was sent.

contains *self changed* as its final statement. Abstract views that have no aspect message update when sent the message *update:* <*object*>, and <*object*> is the abstract view's model. Abstract views update by deleting old subviews then adding new subviews based on the current state of the model.

### 4.7. Primitive Views

Primitive views display data and menus by which the user can issue commands to affect the model. The classes for SIG supplied primitive views are MVCTextView, MVCReadOnlyView, MVCListView, MVCBooleanView, MVCConstantFormView, and MVCCustomView. All primitive views, except constant form views, are pluggable views in that they have at least an aspect message and are assumed to be viewing only a part of some larger model. Primitive views are meant to occur at the leaves of a hierarchy, and, with the exception of constant form view, should have *yourself* as model message. An object that has several instance variables may be the model of several primitive views. For example, the recipe *isListOn→()* in display type *BLTest1* specifies three subviews, each a primitive view: two Boolean views and a list view. In an ID that uses display type *BLTest1*, all three of these views have the same model, the BooleanListTest on which the ID was opened. The model message specified for each subview is *yourself*. This message is sent by the seed abstract view to its model, the BooleanListTest on which the ID was opened, to establish the model for each of the three primitive views. Here this message is sent to the same object. Therefore, all three subviews have the same model, but each subview displays a different part of the model. Figure 4.6 shows the methods in class BooleanListTest. The definition of class BooleanListTest shows that it has instance variables *list* and *value*. In the ID shown in Figures 4.4a and 4.4b, the upper Boolean view is highlighted when the instance variable *value* is true; the lower Boolean view is highlighted when the instance variable *value* is false, and the list view displays the list held in instance variable *list*.

```
Object subclass: #BooleanListTest
    instanceVariableNames: 'list currentlySelectedItem value '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SIG-Tests'


BooleanListTest methodsFor: 'initialize'


initialize
    "Set the list to a example list and the value to false."

    list ←(Array with: 'Line 1'
        with: 'line 2, and its a little longer'
        with: 'third line'
        with: 'line 4 is the longest line yet.  It is a very long line').

    value ←false



BooleanListTest methodsFor: 'recipe adaptor'


isListOn
    "Return true if the ListView should be displayed,
    False otherwise.  The ListView should be displayed
    if the instance variable 'value' is true."

    ↑value



BooleanListTest methodsFor: 'BooleanView adaptor'


value
    "Return a Boolean.  False if my value is nil."

    value notNil
    ifTrue: [↑value]
    ifFalse: [↑false]


value: aBoolean
    "Set my value."

    value ←aBoolean.
    self changed
```

Figure 4.6 Code for Class BooleanListTest

```
BooleanListTest methodsFor: 'ListView adaptor'

list
    "Return the array I am storing."

    ↑list


selection
    "This is the initial selection message."

    ↑currentlySelectedItem


selection: anArrayElement
    "This is the change message."

    currentlySelectedItem ←anArrayElement

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

BooleanListTest class
    instanceVariableNames: ''


BooleanListTest class methodsFor: 'instance creation'


new
    "Create an initialized instance."

    ↑super new initialize
```

Figure 4.6 Code for Class BooleanListTest continued

### 4.7.1. Text View

Text views are instances of MVCTextView, which is a subclass of CodeView. Text views hold text that may be edited with the Smalltalk text editor. Text views require two creation Messages, an aspect message and a change message. The model for text views must return a Text, on receipt of the aspect message, and must expect a Text on receipt of the change message. For the demonstration class BTN, the aspect and change messages are *textCopy* and

*accept:*, respectively.[8]

### 4.7.2. Read-Only View

Read-only views are instances of MVCReadOnlyView, which is a subclass of MVCTextView. Read-only views act just like text views, except they have no editing menu. They exist so text can be presented in a view that scrolls. The view that appears at the top of Figure 2.1 is a read-only view. This view can be the height of one line of text and still be effective, even with a very long Prolog literal, because it allows scrolling. Since read-only views have no *accept* menu item, they require no change message. They must have an aspect message, however, so the model can request that a read-only view be updated and the view can obtain the text to display.

### 4.7.3. List View

List views are instances of MVCListView, which is a subclass of SelectionInListView. List views behave like the list views in the system browser; they display a list of items, one line per item. If an item's description is longer than one line, the description is truncated to one line. At most one item at a time is selected, and the model keeps track of the selection. List views require only two creation messages, the aspect message and the change message. Unlike SelectionInListViews, when the aspect message is sent to the model of a list view, it is expected to return an array of items in the list; so it functions as both the aspect message and the list message of SelectionInListViews. This modification makes list views and text views behave consistently. A list view sends its change message to the model when a new item is selected in the view. The print items parameter, one item parameter, and initial selection message function the same as for SelectionInListViews.

Figure 4.5 shows that display type *BLTest1* specifies that the aspect, change, and initial selection messages for the list view are *list*, *selection:*, and *selection*, respectively. Figure 4.6

[8]See the protocol *text view adaptor* in Appendix A.

shows the methods for these messages under the protocol *ListViewAdaptor*. The aspect message, *list*, returns the list stored in the model. The change message, *selection:*, is sent to the model with an argument that is the selected item in the list. The model stores the selected item in the instance variable *currentlySelectedItem*. The initial selection message, *selection*, returns the currently selected list item to the view.

### 4.7.4. Boolean View

Boolean views are instances of MVCBooleanView, which is a subclass of BooleanView. The functionality of Boolean views is the same as that of system-supplied BooleanViews with minor modifications. Boolean views require the same creation messages as BooleanViews: aspect and change messages, and label and value parameters. As we have seen, the display type *BLTest1* calls for two Boolean views with labels *list on* and *list off*, one of which is always black and the other always white. A pair of Boolean views often act as a switch in this way. When the user indicates one Boolean view with the mouse that Boolean view is highlighted (displayed with a black background) and the associated Boolean view is displayed normally (white background). In the BooleanListTest example, the list view is displayed when the Boolean view with label *list on* is highlighted, and not displayed when the Boolean view with label *list off* is highlighted. (See Figures 4.4a and 4.4b.) To achieve this pairing, both Boolean views have the same aspect message, *value*, and and the same change message, *value:*. The value parameter is true for the *list on* Boolean view, and false for the *list off* Boolean view. After receiving and storing a new Boolean value, the method for the change message, *value:*, executes *self changed*, as shown in Figure 4.6, which ensures that whenever one of the Boolean views updates the model both views are redisplayed.

### 4.7.5. Constant Form View

Constant form views are instances of MVCConstantFormView, which is a subclass of FormView. A constant form view is the only kind of primitive view that is not pluggable, so it

requires no creation messages. The models for constant form views must be instances of Form or OpaqueForm. These forms are stored in the specification, and are not some subpart of the model. Constant form views use the display mechanism of FormViews, which resizes the form to fit in the view. Actually, the FormView obtains a new form of the appropriate size, so the form in the specification is left untouched. The display type *withArrows*, illustrated in Figure 4.7, uses constant form views to display arrows between nodes in a binary tree.

### 4.7.6. Custom View

Custom views are instances of MVCCustomView, which is a subclass of MVCView, a subclass of View. Custom views allow users to implement their own display routines in the model. Custom views are pluggable. They require an aspect message selector so the model can selectively update them. Whenever a custom view receives the message *update:* <*symbol*>, and <*symbol*> matches the aspect message for the custom view, the custom view sends a *display message* to its model. The model itself updates the screen in response to the display message. Like the other creation messages, the display message is stored in the specification for an MVC triple, but not with the other creation messages. A custom view never informs its model of a change in the view, obviating the need for a change message. The display type *customArrows* for class BTN, explained in Chapter 6, uses custom views for its arrows.

### 4.7.7. User Defined Primitives

Users can add their own primitive views to SIG. A user might want to create a new primitive view to give new functionality to the red button or keyboard. If a new primitive view is needed, the new view class needs the following characteristics to fit into the SIG system. It must have an instance variable *mvcSpec* and implement the instance messages *on:mvcSpec:*, *release*, *mvcSpec*, *mvcSpec:*, and the class messages *minimumSize* and *standardMenu*. The messages *on:mvcSpec:* and *release* are sent by an abstract view to create and release a primitive view. The messages *mvcSpec* and *mvcSpec:* allow other parts of the SIG system to access the

*withArrows*

*#isLeaf→()*

```
MVCTextView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0@0 corner: 1@1
    creationMessages: Dictionary (#aspect→#textCopy #change→#accept: )
    menu: ActionMenu
        labels: 'again
            undo
            copy
            cut
            paste
            do it
            print it
            accept
            cancel
            add left
            add right
            delete'
        lines: #(2 5 7 9 )
        selectors: #(#again #undo #copySelection #cut #paste #doIt #printIt
            #accept #cancel #addLeft #addRight #delete )
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
    displayMessage: nil
    controllerClass: nil
```

*#isLeftChildNil→()*

```
MVCTextView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0.25@0 corner: 0.75@0.2
    creationMessages: Dictionary (#aspect→#textCopy #change→#accept: )
    menu: ActionMenu
        labels: 'again
            undo
            copy
            cut
            paste
            do it
            print it
            accept
```

Figure 4.7 Display Type *withArrows*

```
                cancel
                add left
                add right
                delete'
            lines: #(2 5 7 9 )
            selectors: #(#again #undo #copySelection #cut #paste #doIt #printIt
                #accept #cancel #addLeft #addRight #delete )
        subject: nil
        leftBorder: 1
        rightBorder: 1
        topBorder: 1
        bottomBorder: 1
        displayMessage: nil
        controllerClass: nil

    MVCConstantFormView
        modelMessage: nil
        modelArgument: an OpaqueForm
        rectangleMessage: nil
        rectangleArgument: 0.5@0.2 corner: 0.75@0.3
        creationMessages: nil
        menu: nil
        subject: nil
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil

    MVCAbstractView
        modelMessage: #rightChild
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0.5@0.3 corner: 1@1
        creationMessages: nil
        menu: nil
        subject: 'withArrows'
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil

#isRightChildNil→()

    MVCTextView
        modelMessage: #yourself
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0.25@0 corner: 0.75@0.2
```

Figure 4.7 Display Type *withArrows* continued

```
    creationMessages: Dictionary (#aspect→#textCopy #change→#accept: )
    menu: ActionMenu
        labels: 'again
            undo
            copy
            cut
            paste
            do it
            print it
            accept
            cancel
            add left
            add right
            delete'
        lines: #(2 5 7 9 )
        selectors: #(#again #undo #copySelection #cut #paste #doIt #printIt
            #accept #cancel #addLeft #addRight #delete )
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
    displayMessage: nil
    controllerClass: nil

MVCConstantFormView
    modelMessage: nil
    modelArgument: an OpaqueForm
    rectangleMessage: nil
    rectangleArgument: 0.25@0.2 corner: 0.5@0.3
    creationMessages: nil
    menu: nil
    subject: nil
    leftBorder: 0
    rightBorder: 0
    topBorder: 0
    bottomBorder: 0
    displayMessage: nil
    controllerClass: nil

MVCAbstractView
    modelMessage: #leftChild
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0@0.3 corner: 0.5@1
    creationMessages: nil
    menu: nil
    subject: 'withArrows'
    leftBorder: 0
    rightBorder: 0
    topBorder: 0
    bottomBorder: 0
```

Figure 4.7 Display Type *withArrows* continued

```
        displayMessage: nil
        controllerClass: nil

#notNil→()

    MVCTextView
        modelMessage: #yourself
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0.25@0 corner: 0.75@0.2
        creationMessages: Dictionary (#aspect→#textCopy #change→#accept: )
        menu: ActionMenu
            labels: 'again
                undo
                copy
                cut
                paste
                do it
                print it
                accept
                cancel
                add left
                add right
                delete'
            lines: #(2 5 7 9 )
            selectors: #(#again #undo #copySelection #cut #paste #doIt #printIt
                #accept #cancel #addLeft #addRight #delete )
        subject: nil
        leftBorder: 1
        rightBorder: 1
        topBorder: 1
        bottomBorder: 1
        displayMessage: nil
        controllerClass: nil

    MVCConstantFormView
        modelMessage: nil
        modelArgument: an OpaqueForm
        rectangleMessage: nil
        rectangleArgument: 0.25@0.2 corner: 0.5@0.3
        creationMessages: nil
        menu: nil
        subject: nil
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil

    MVCAbstractView
        modelMessage: #leftChild
```

Figure 4.7 Display Type *withArrows* continued

```
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0@0.3 corner: 0.5@1
        creationMessages: nil
        menu: nil
        subject: 'withArrows'
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil

    MVCConstantFormView
        modelMessage: nil
        modelArgument: an OpaqueForm
        rectangleMessage: nil
        rectangleArgument: 0.5@0.2 corner: 0.75@0.3
        creationMessages: nil
        menu: nil
        subject: nil
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil

    MVCAbstractView
        modelMessage: #rightChild
        modelArgument: nil
        rectangleMessage: nil
        rectangleArgument: 0.5@0.3 corner: 1@1
        creationMessages: nil
        menu: nil
        subject: 'withArrows'
        leftBorder: 0
        rightBorder: 0
        topBorder: 0
        bottomBorder: 0
        displayMessage: nil
        controllerClass: nil
```

Figure 4.7 Display Type *withArrows* continued

MVCSpecification stored by a primitive view. The message *minimumSize* returns the minimum screen area a view may occupy. The message *standardMenu* returns a menu containing the messages implemented by the default controller for a view.

### 4.8. Structure of Specifications

Instances of the class MVCSpecification represent specifications. A specification holds the parameters necessary for an abstract view to add an MVC triple to the current ID. Specifying an MVC triple means supplying values for the instance variables of an MVCSpecification. As shown in Figure 4.5, the display type *BLTest1* contains five specifications: three for recipe *isListOn→()* and two for recipe *notNil→()*. The following discussion pertains to the specifications for recipe *isListOn→()*. These three specifications are called simply the first, second, and third specification. The names and expected values for the instance variables of class MVCSpecification are given below.

viewClass [Class]

The view component of the MVC triple specified is an instance of Class. The following are the current legal classes: MVCAbstractView, MVCBooleanView, MVCConstantFormView, MVCCustomView, MVCListView, MVCReadOnlyView, and MVCTextView. The value of *viewClass* for the first and second specifications is MVCBooleanView. The value of *viewClass* for the third specification is MVCListView.

modelMessage [nil | Symbol]

Symbol is the model message that an abstract view sends to its model. The object returned becomes the model of the subview specified by this specification. All views except abstract and constant form views use *yourself* as their model message. The value of *modelMessage* for all three example specifications is *yourself*.

modelArgument [nil | Array | OpaqueForm]

Generally this variable is nil. If the model message has arguments, this variable is an array containing the arguments. The display type *PLOriginal*, for class PrologLiteral, uses *modelArgument* in this manner. If the value of *viewClass* is MVCConstantFormView, this variable may contain that Form.

rectangleMessage [nil | Symbol]

This variable is only used if the model knows how to calculate rectangles; normally it is nil. The display type *PLOriginal* uses a rectangle message. Each Prolog literal calculates the height of the rectangle needed to display its functor based on the number of arguments that Prolog literal has. If the variable *rectangleMessage* is not nil, the message sent must return a relative rectangle. (Relative rectangles are explained in Section 6.1.) Instances of class BooleanListTest do not care to calculate rectangles for their views, so the value of *rectangleMessage* is nil for all three example specifications.

rectangleArgument [Rectangle | Array]

If *rectangleMessage* is nil, this variable must be a relative rectangle, which is used to hold the subview being added. If rectangleMessage is not nil, *rectangleArgument* must be an array of arguments. In this case, the selector found in *rectangleMessage* and the arguments are sent to the model to return a relative rectangle. In the example, the first specification has *0@0 corner: 0.25@0.15* as the value for *rectangleArgument.* This means that the origin (top left corner) of the specified subview, the Boolean view *list on*, is at the origin of the seed abstract view, as shown in Figure 4.4a.

creationMessages [nil | Dictionary]

All MVC views except top views and constant form views have creation messages. Abstract views have only an aspect message. When an abstract view has an aspect message, the model message should be *yourself*; otherwise, the model message should return a subpart of the current model. In the BTN example of Chapter 6, the model messages for the abstract views are *leftChild* or *rightChild,* which return the left or right child, respectively, of a binary tree node.

The variable *creationMessages* usually contains a dictionary, which is a set of key-value pairs (associations). The kind of message or parameter: aspect, change, label, value, print items, one item, or initial selection, is the key and a message selector, Boolean value, or

string is the value. The keys that are present depend on the class specified in *viewClass*. The values associated with keys aspect, change, and initial selection are message selectors. The value associated with key label is a string. The values associated with value, print items, and one item must be Booleans, as explained in Chapter 3.

In the example, all three specifications have a dictionary as the value of *creationMessages*. The dictionary in the first specification contains the creation messages necessary for a Boolean view. The label for this Boolean view is *list on* and the value *true* is sent to the model with the change message. The second specification has a similar dictionary. The third specification has a dictionary whose keys are those required by list views. The aspect, change, and initial selection messages are: *list*, *selection:*, and *selection*. The print items and one item parameters are both false, indicating that the model maintains a list of strings, and that the list may have more than one item.

menu [nil | ActionMenu]

If the view described by a specification should have no menu, the instance variable *menu* is nil, otherwise it must be an ActionMenu. Text views have a standard menu with the normal text manipulation commands. Most other views have an empty standard menu. When editing a specification, whenever the variable *viewClass* is changed, the instance variable *menu* is automatically filled with the standard menu for the view class given in *viewClass*. The standard menu for any MVC view, except constant form view, may be augmented by messages that should appear in the yellow button menu. Any items added to the menu must have corresponding methods in the model. An example of augmenting the menu for text views is given in Chapter 6.

subject [nil | String]

The subject of an English sentence is a noun, a descriptive entity. The instance variable *subject* contains the display type that provides the descriptive information used by an abstract view to propagate an ID. This variable is nil unless the view being specified is an

abstract view. If the variable *viewClass* specifies an abstract view, the variable *subject* must be the name of a display type. This display type is the description for a sub-ID. As explained above, an abstract view uses the condition messages in the recipes from this display type to determine the specifications for subviews of the current view. The display type *customArrows* for class BTN contains specifications for abstract views.

Border [Integer]

There are actually four variables here, *leftBorder*, *rightBorder*, *topBorder*, and *bottomBorder*. Each holds an integer that specifies the width in pixels of the corresponding border for this view. Borders are always black. If the specified integer is 0, no border is displayed. The example specifications all specify all four borders to have width 1.

displayMessage [nil | Symbol]

This variable is only used by custom views, and contains the display message. The custom view described by this specification, sends the display message to its model any time the custom view receives the message *displayView*. When the model receives the display message, it can update the display as it sees fit. This message requests customized drawings directly on the screen, so the model usually needs to know about the view that sent this message so it knows where on the screen to draw. A display message is used in the display type *customArrows* described in Chapter 6.

controllerClass [nil | Class]

This variable is almost always nil, as all SIG views have default controller classes. If it is not nil, it must be a class whose instances are valid controllers for the view class specified in *viewClass*.

### 4.9. Error Handling - Too-small Views

In an ID, such as on a binary tree or a Prolog literal, in which views are being added and deleted, it is possible to have too little space on the screen to display a view. Both the binary

tree and Prolog literal examples use text views. Text views must be big enough to display at least some text, about 30 x 20 pixels, or an error will occur. All other MVC views also have a minimum size, with the default being 10 x 10 pixels. After an abstract view has determined the rectangle in which a new view is to be displayed, it checks this rectangle against the minimum size of the view to be added. When the rectangle is smaller than this minimum size, a too-small view is added instead. Too-small views have a dark gray background and store the specification for the view they are replacing. A controller for a too-small view displays a yellow button menu with one message, *expand*. When this menu item is selected, a new top view is spawned using the specification stored by the too-small view. The model of the too-small view becomes the model of the new top view. Figure 4.8 shows an ID on a binary tree having too-small views, and a new top view spawned from the too-small view. There are actually three too-small views, replacing two text views and a custom view, as shown in the spawned ID.

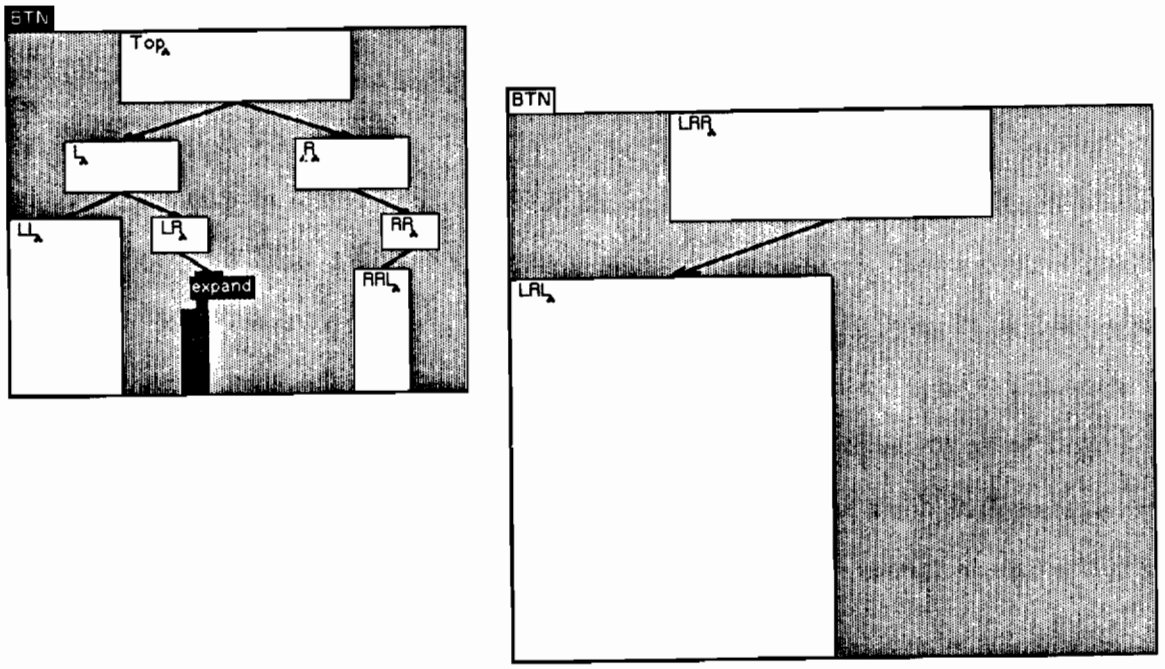


Figure 4.8 Expansion of an MVCTooSmallView

# 5. Using SIG

## 5.1. The Display Type Dictionary

A dictionary collects all display types in SIG. This dictionary is globally available in a variable called *DisplayBuilder*. *DisplayBuilder*, like the system dictionary, *Smalltalk*, is the sole instance of its class, DisplayGenerator. Besides storing display types, *DisplayBuilder* serves two other functions in the SIG system:

(1) it invokes the display type editor and

(2) it starts IDs on instances of classes for which display types have been defined.

## 5.2. Display Type Editor

To open a display type editor, execute *DisplayBuilder edit*, which creates a *new* ID with six visible subviews. Figures 5.1a and 5.1b show two display type editors, each viewing the same display type. The editor in the top figure shows the initial configuration, which allows editing. The editor in the bottom figure is in display-only mode, which is useful for scanning display types. The user can control the mode of a display type editor. The menu items available within each view are also shown. Figure 5.2 shows a schematic diagram of the visible subviews in a display type editor. Each subview is numbered, for reference purposes, and labeled, to denote the kind of data displayed. Views 1 through 5 are list views. In all of these list views, at various times, the user may be prompted for a name of some kind. Such prompts are given to obtain the information necessary for SIG to carry out some operation, e.g., adding a new display type to *DisplayBuilder*.

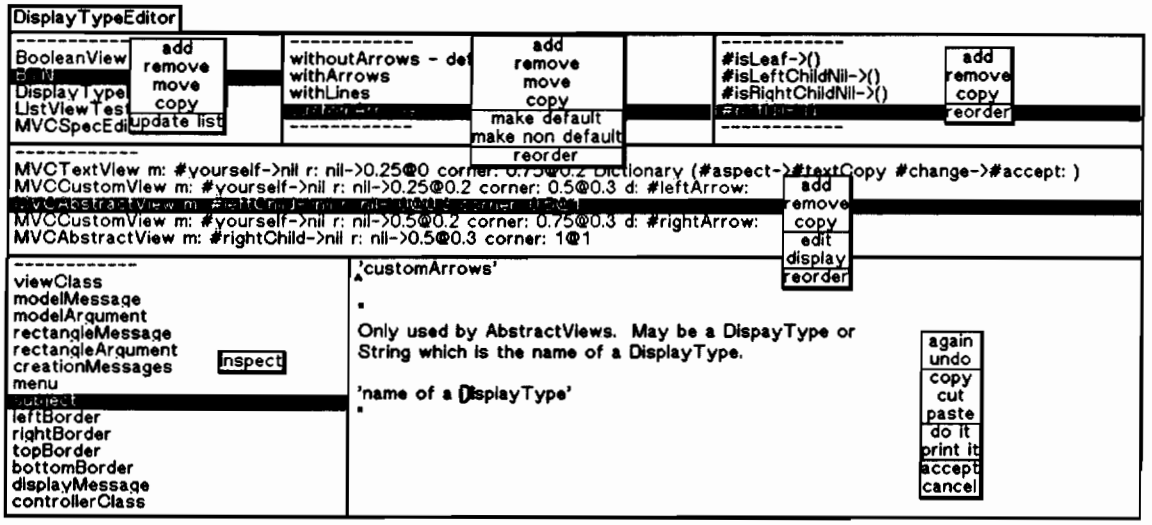


Figure 5.1a Display Type Editor in Edit Mode

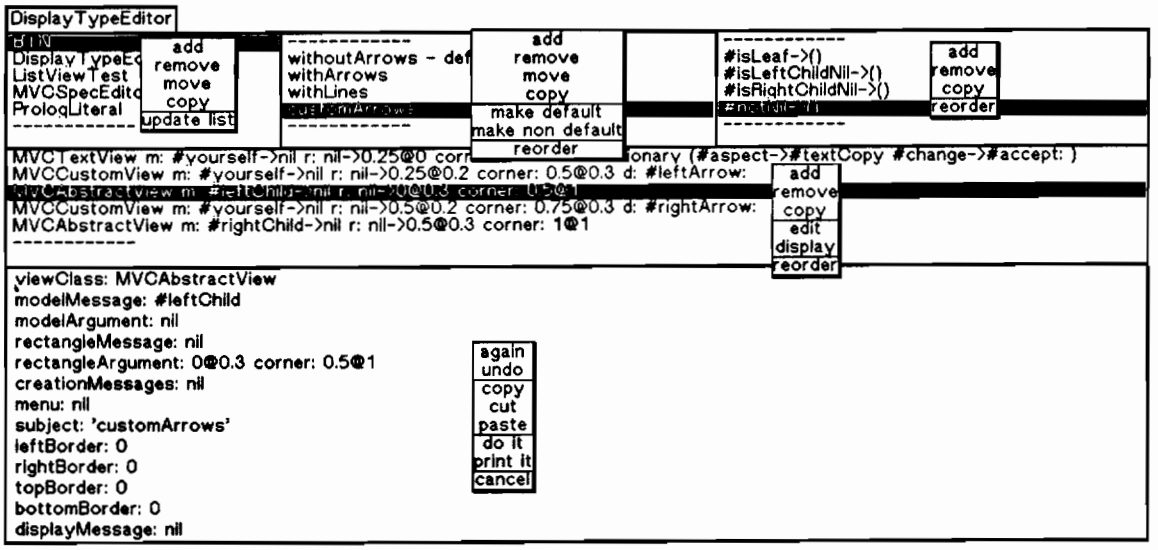


Figure 5.1b Display Type Editor in Display Mode

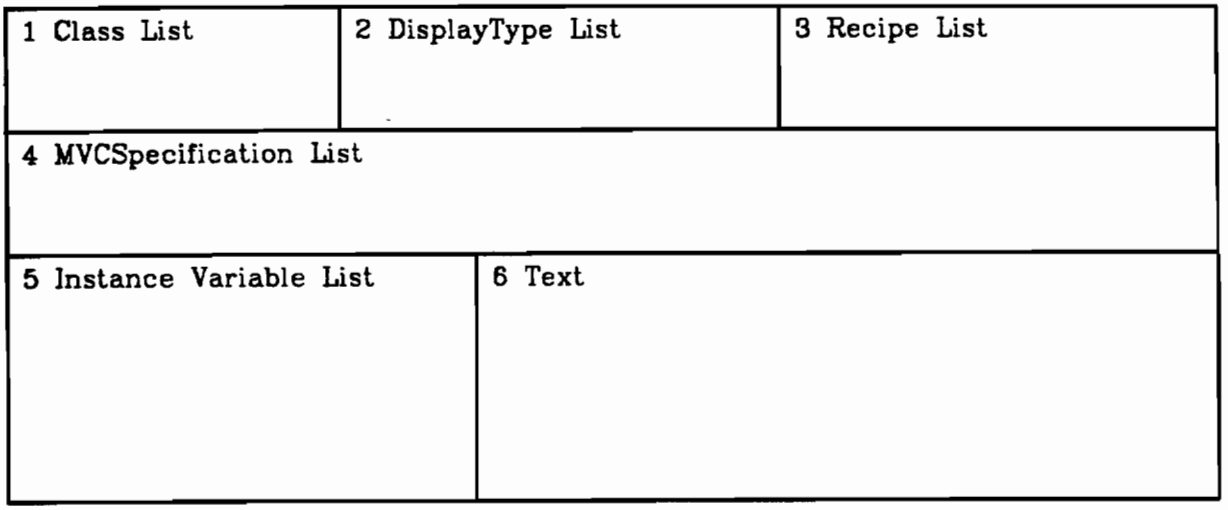


Figure 5.2 View Scheme of the Display Type Editor

The views that comprise a display type editor have dependence relationships that are similar to the relationships between the views in the Smalltalk supplied system browser. View 1 displays the list of classes for which display types have been defined; view 2, the display types defined for the class selected in view 1; view 3, the recipes for the display type selected in view 2; and view 4, the specifications for the recipe selected in view 3. View 5 displays a list of the instance variables of the specification selected in view 4. All specifications have the same instance variables, so view 5 displays a list containing the following items: *viewClass*, *modelMessage*, *modelArgument*, *rectangleMessage*, *rectangleArgument*, *creationMessages*, *menu*, *subject*, *leftBorder*, *rightBorder*, *topBorder*, *bottomBorder*, *displayMessage*, and *controllerClass*, as shown in Figure 5.1a. View 6 displays the value of the instance variable selected in view 5.

### 5.2.1. Class List View

In display type editors, the class list view contains a list of classes for which display types exist. The editors shown in Figures 5.1a and 5.1b show classes BTN and ListViewTest, among others. There are five items on the yellow button menu for the class list view: *add*, *remove*, *move*, *copy*, and *update list*. Of these, *add* and *update list* are meaningful whether or not there is

a current selection, while a *remove*, *move*, or *copy* operation requires that a class is selected.

The message *add* prompts for the name of a class to be added to *DisplayBuilder* and to the class list view. If the user adds a class that is already in the class list, SIG asks, whether the display types currently associated with this class should be discarded. Menu item *remove* removes the selected class from *DisplayBuilder*. If the selected class has display types, SIG asks whether or not to to delete the display types for this class. The item *move* acts as a rename operator. SIG prompts for a class name. The display types associated with the currently selected class are associated with the class given in response to the prompt. The currently selected class is then removed from *DisplayBuilder*. The *copy* operation is the same as *move*, except the selected class is not removed from *DisplayBuilder*. Instead, a copy is made of each display type in the selected class and these copies are associated with the specified class.

The last menu item, *update list*, is used when the class list appearing is not a correct representation of the actual list of classes for which there are display types. This discrepancy may arise when a class in the class list, say Test, is modified through a system browser. Figure 5.3a shows the display type editor after such a change. Test is the class selected in the class list view. However, *DisplayBuilder*, no longer references class Test, but an obsolete version of class Test. Figure 5.3b shows the display type editor after an *update list* operation has been done.

### 5.2.2. Display Type List View

The display type list view contains the list of display types for the class selected in the class list view. Figures 5.1a and b show a display type editor editing display types for the example class BTN. The display type list view has the following yellow button menu items: *add*, *remove*, *move*, *copy*, *make default*, *make non-default*, *reorder*. There need be no current selection in the display type list view when the operations *add* and *reorder* are initiated. The other operations must have a selection upon which to operate.

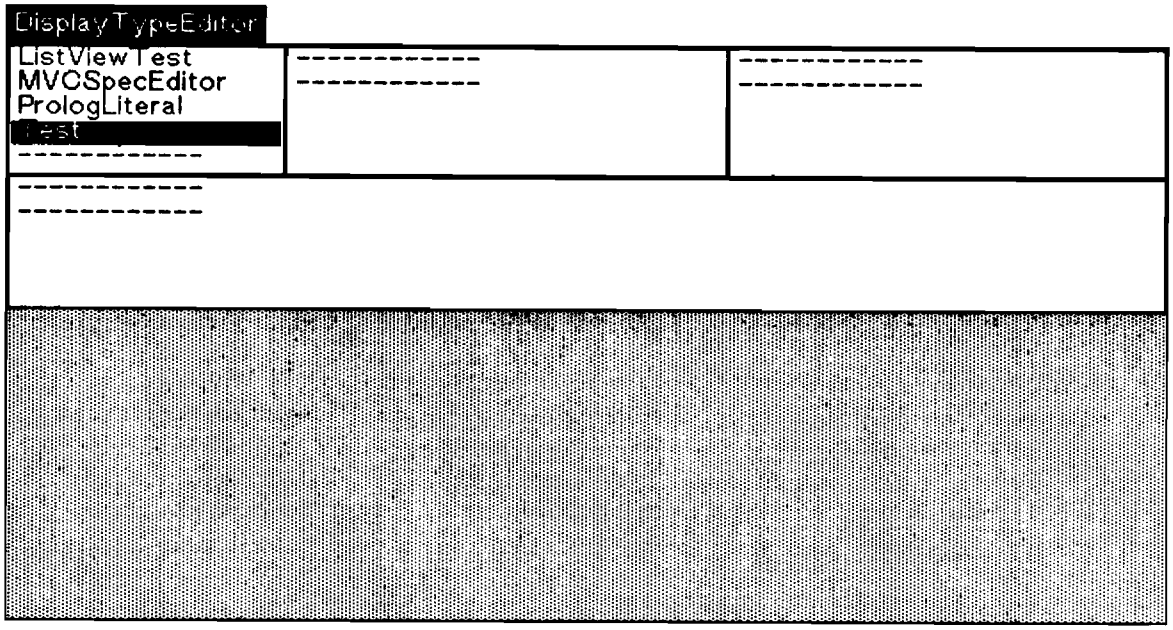


Figure 5.3a An Incorrect Class List View

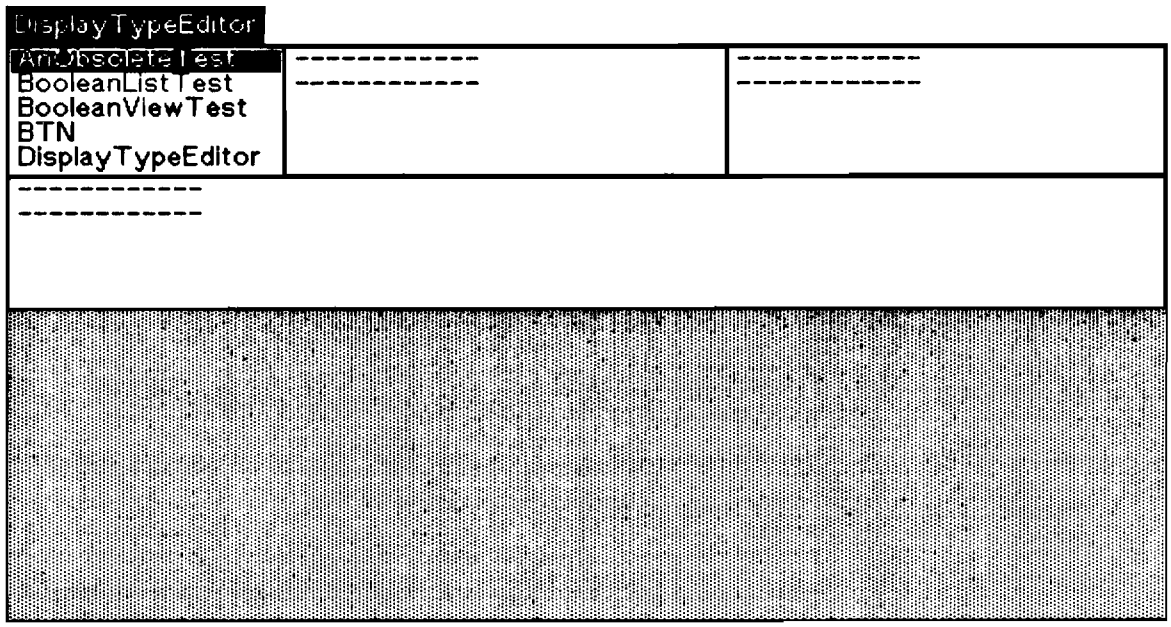


Figure 5.3b After the *update list* Operation

The *add* message is similar to the class list view add; the differences are that it prompts for a display type name rather than a class name. The *remove* message for this view operates

as the class list view *remove*. The *move* and *copy* operations also function in a similar manner, except that they allow the user to move or copy a display type to a new class as well as to a new display type within the selected class. Menu items *make default* and *make non-default* allow the user to toggle the *default* switch for the selected display type. The operation *make default* makes the selected display type the default display type for the selected class. The operation *make non-default* unsets the *default* switch for the selected display type. The *reorder* message is used to reorder the display types in the display type list view.

### 5.2.3. Recipe List View

The recipe list view, in the display type editor, shows a list of all the recipes for the display type currently selected in the display type view. Each item in the list represents the condition message for a recipe. These condition messages are displayed in the format *#selector→(list of arguments)*. For example, the first recipe for display type *PLOriginal* for class Prolog literal has condition message *#argSize→(0)*. When the condition message has no arguments, the display has the form: *#selector→()*, as in Figures 5.1a and b.

The menu messages available in the recipe list view are: *add*, *remove*, *copy*, and *reorder*. All messages behave much the same as their counterparts in the display type list view. Unlike the *copy* operation in the display type list view, the recipe list view *copy* will not copy a recipe to a display type other than the one currently selected.

### 5.2.4. Specification List View

Items in the specification list view represent MVC triples that are added to an ID whenever the currently selected recipe matches the model. The representations shown in this list are summaries of specifications. For example, the selected item in the specification list view of Figure 5.1b contains the text

MVCAbstractView m: #leftChild→nil r: nil→0@0.3 corner: 0.5@1.

The first item listed is the value of *viewClass*. The *m:* indicates that the next selector→argument-list pair makes up the model message. The *r:* indicates that the next message is the rectangle message. Other items that may appear in a summary are creation messages and the display message, if they exist.

The yellow button menu for the specification list view is: *add*, *remove*, *copy*, *edit*, *display*, and *reorder*. When the *add* operation is used to add a new specification to the list, no prompt is issued, but the summary for this specification appears in the list as *nil m: nil→nil r: nil→nil*. Understandably, *copy* does not prompt either. Other than these differences, *add*, *remove*, *copy*, and *reorder* operate as they do in the recipe list view. The menu items *edit* and *display* are mode selectors. The display type editor in Figure 5.1a is in edit mode, which is the default, and the display type editor in Figure 5.1b is in display mode.

### 5.2.5. Views on an Individual Specification

Views 5 and 6 in Figure 5.1a show an editor on a single specification. In Figure 5.1b, these two views are replaced by a single view that may only display, not edit, a specification.

#### 5.2.5.1. Edit Mode

The two bottom views of Figure 5.1a resemble an inspector, and they behave almost like an inspector on the specification selected in the specification list view. There are two major differences: the text that appears in the right view contains a template for producing a new value for the instance variable selected in the left view, and the right view has no *do it* context. In this respect, the text view in a display type editor is more like a workspace than an inspector. Notice also that *self* does not appear in view 5 of the display type editor, as it does in the left view of normal inspectors.

View 5 in Figure 5.1a shows a list of the instance variables in a specification. Selecting any item in the list causes the text view to the right to be updated. This view has one menu message, *inspect*, that spawns an inspector on the currently selected instance variable. This

operation is particularly useful when editing the instance variables *creationMessages* and *menu*.

View 6 of Figure 5.1a is a text view displaying the value of the instance variable, *subject*, selected in view 5. Whenever a new instance variable is selected in view 5, view 6 is updated. The yellow button menu for this view is *again*, *undo*, *copy*, *cut*, *paste*, *do it*, *print it*, *accept*, and *cancel*. These messages operate as they do in the system browser with the exception of *accept*. When *accept* is selected, the text in view 6 is compiled and executed. The object returned from this execution becomes the new value of the instance variable currently selected in view 5.

In addition to the value of the instance variable selected in view 5 of the display type editor, view 6 displays a template that can be instantiated to reset that value. View 6 in Figure 5.1a, contains the following text:

```
'customArrows'

"
Only used by AbstractViews. May be a DisplayType or
String which is the name of a DisplayType.

'name of a DisplayType'
"
```

The string 'customArrows' is the current value of the instance variable *subject*. The template in this case is *'name of a DisplayType'*. The current value of the instance variable selected in view 5 always appears at the top of view 6 with the template for the selected instance variable below this value enclosed in double quotes.

Templates consist of two parts, a comment and an executable text. The comment explains something about the instance variable's meaning and expected value. Below the comment appears an executable portion of text. It is separated from the comment by a blank line. This text has the form needed to create an object of the correct type for the currently selected instance variable. One way to use the template is to edit the sample values it contains, paste in the edited version of the template over the current template, and *accept*. Figure 5.4 shows this process for the deletion of the change message for a text view. The text manipulations

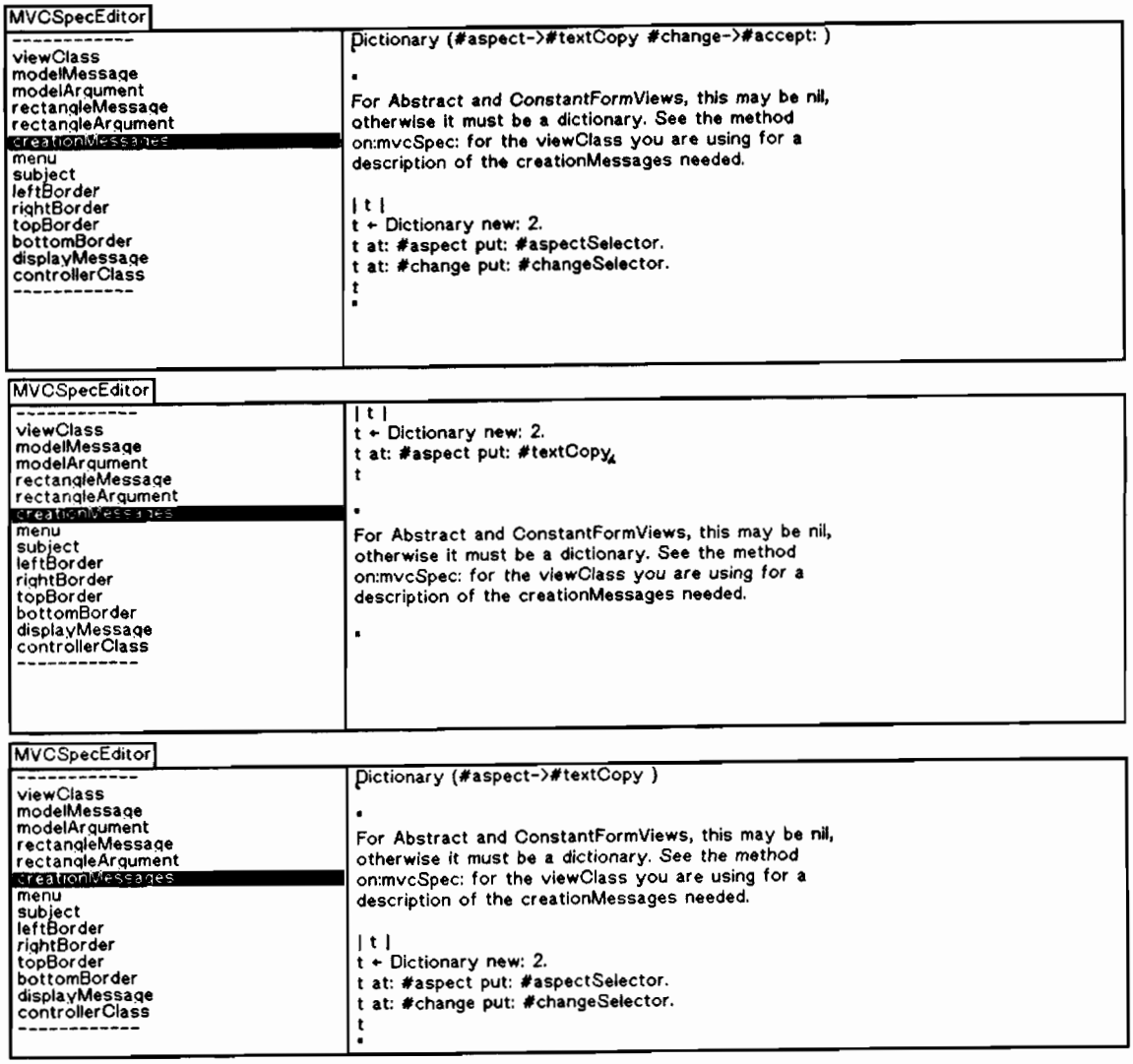


Figure 5.4 Using Templates

explained above are necessary to ensure that executable text appears in the text view at the time of the *accept*.

### 5.2.5.2. Display Mode

Figure 5.1b shows a display type editor in display mode. In this mode the lowermost view shows all the instance variables and their values for the selected MVCSpecification. The menu messages are the same as the text view in edit mode, except *accept* is absent. The major use

for display mode is seeing all the instance variables in the selected specification at one time.

#### 5.2.5.3. Checking Validity

Changes to a specification can render it invalid. Hence, when the user attempts to leave the selected specification, a check is done to ensure that it meets certain integrity constraints. If anything looks amiss, the user is asked to confirm the value of the offending instance variable(s). The confirmers that appear are self explanatory, for example,

> You have neither a modelMessage nor a modelArgument.
> Proceed anyway?

## 5.3. Model Messages

In creating a display type for a class, several messages are specified to which the model must respond. These messages can be found using the display type editor. All the messages that must be implemented fall into one of the following categories: condition, model, rectangle, creation, menu, or display. To find these messages for a display type, a user first selects the appropriate class and display type in a display type editor. Each of the messages present in the recipe list must be implemented. After determining what messages must be implemented, the display type editor may be changed to display mode. For each recipe, the user selects each specification in turn. The instance variables *modelMessage*, *rectangleMessage*, *creationMessages*, *menu*, and *displayMessage* can contain message selectors that must be implemented.

## 5.4. Opening an ID on an Instance of a Class

After a display type has been created for a class, *DisplayBuilder* can open an ID using this display type. Two messages exist for opening an ID: *open:* <*anObject*> and *open:* <*anObject*> *using:* <*aDisplayTypeName*>. The first message takes as its argument any object whose class has a display type, and uses the default display type for the class. *DisplayBuilder* opens an ID on <*anObject*> using the default display type it found. For example, to start an ID on

a binary tree, an instance of class BTN, execute *DisplayBuilder open: (BTN new)*. The result of this execution is shown in Figure 5.5. A new BTN has no children, i.e., it is a leaf, so the display has only one view, a text view that fills the entire display area. The second message is used for opening IDs using a display type that is not the default.

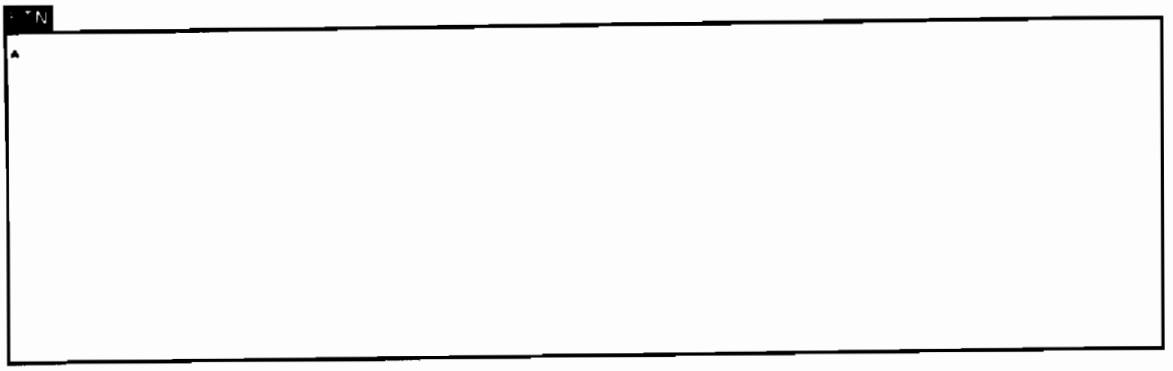

Figure 5.5 A New BTN

# 6. Building an ID

Figure 4.2 shows an ID on a binary tree specified by the display type *withoutArrows*. This chapter explains how to build a similar ID for binary trees specified by a new display type *customArrows*. Figure 6.1 shows the ID that results from executing *DisplayBuilder open: BTN new using: 'customArrows'*. IDs specified by the display type *customArrows* display binary trees by displaying the label of each node in a text view. On the screen below and to the left and right of this text view are abstract views that recursively display the left and right children of the node. Custom views display arrows connecting the text view and the left and right children. In Figure 6.1, the yellow button is down and the user is just about to add a left child to the (currently unlabeled) top node. After adding and labeling a few nodes, the ID appears as shown in Figure 6.2.

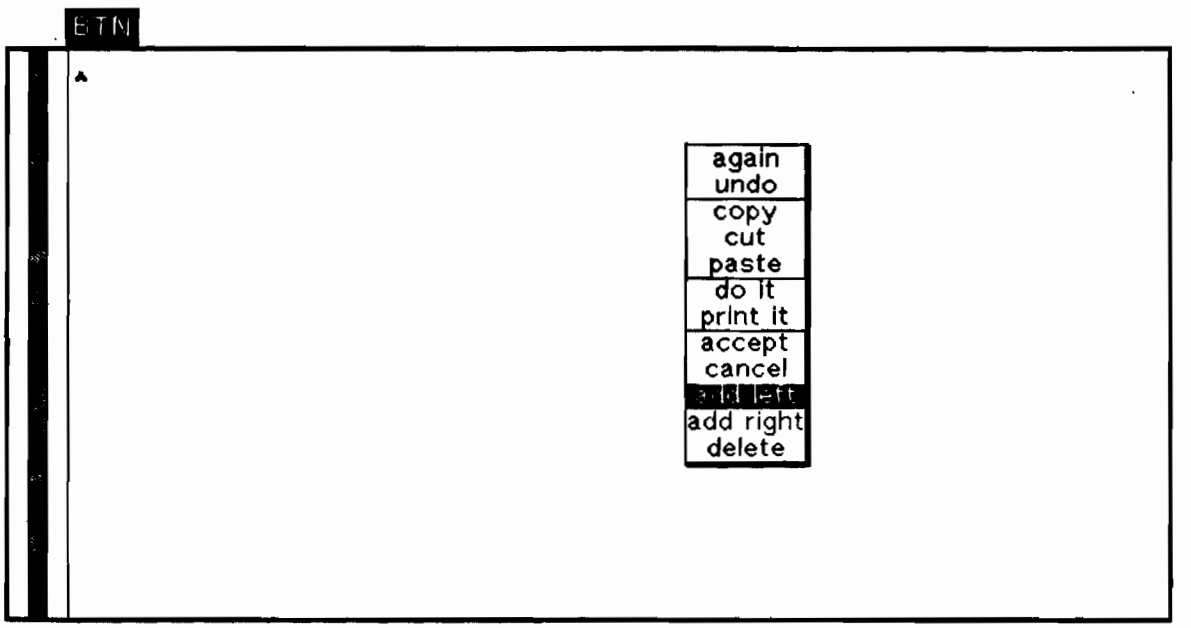


**Figure 6.1 Opening the ID**

# 6. Building an ID

Figure 4.2 shows an ID on a binary tree specified by the display type *withoutArrows*. This chapter explains how to build a similar ID for binary trees specified by a new display type *customArrows*. Figure 6.1 shows the ID that results from executing *DisplayBuilder open: BTN new using: 'customArrows'*. IDs specified by the display type *customArrows* display binary trees by displaying the label of each node in a text view. On the screen below and to the left and right of this text view are abstract views that recursively display the left and right children of the node. Custom views display arrows connecting the text view and the left and right children. In Figure 6.1, the yellow button is down and the user is just about to add a left child to the (currently unlabeled) top node. After adding and labeling a few nodes, the ID appears as shown in Figure 6.2.

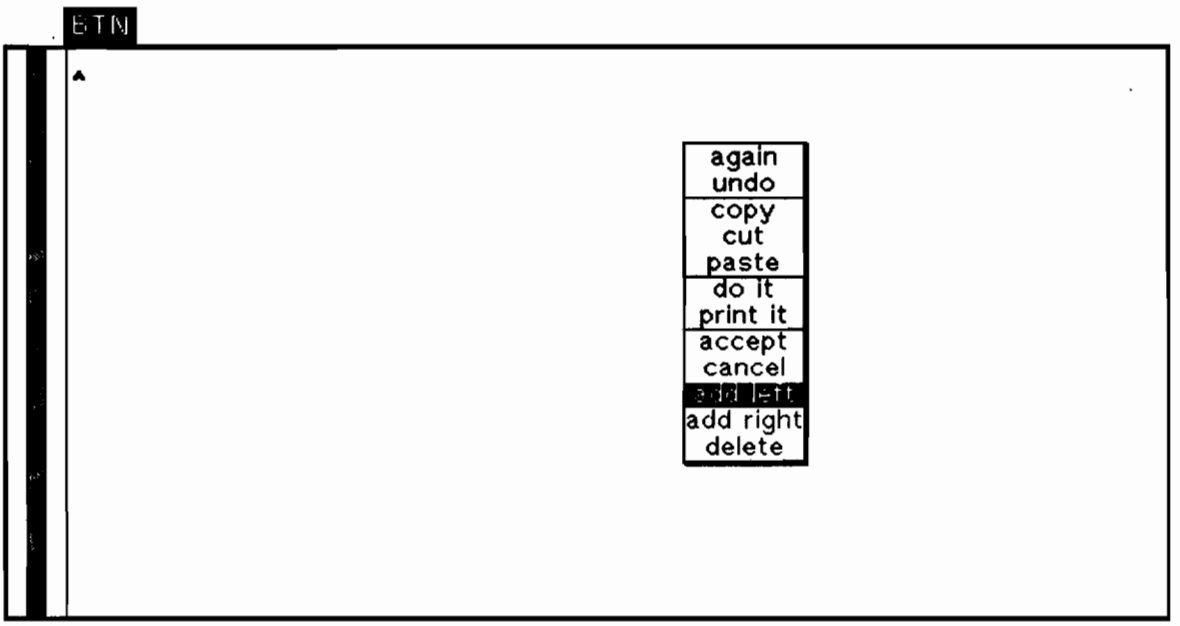


Figure 6.1 Opening the ID

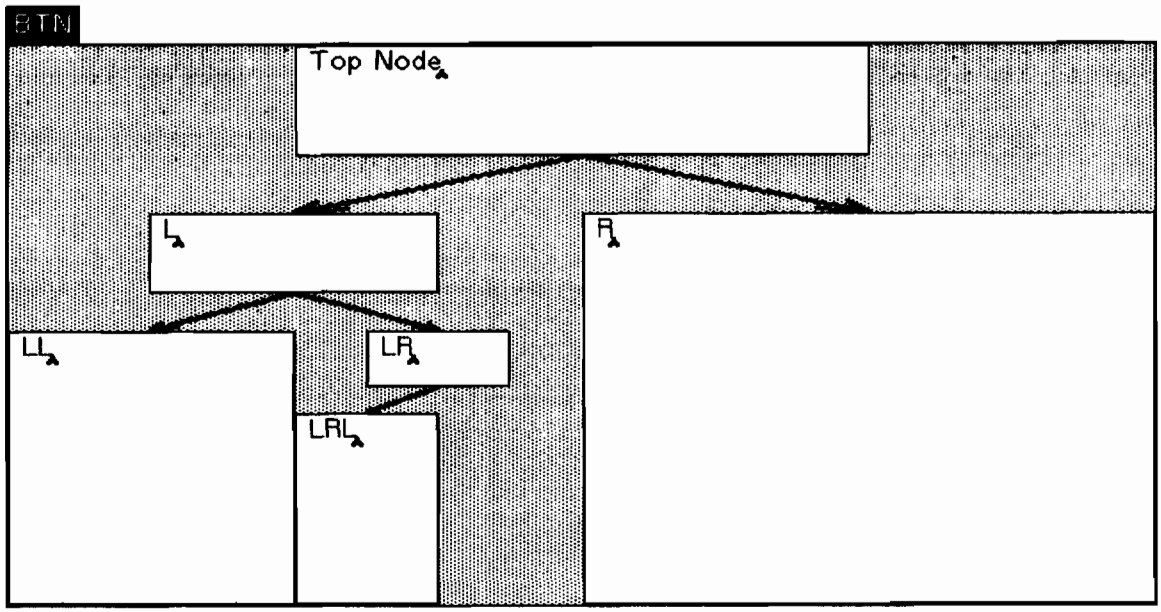


Figure 6.2 After Several Additions

The procedure for building IDs like the one shown in Figure 6.1 and 6.2 follows. First one must create the class whose instances will be the models of IDs. After creating a class, a display type is created using the display type editor. Figure 6.3 shows a display type editor after *customArrows* has been added to the list of display types associated with class BTN, but before any recipes have been given for the new display type. Before IDs using the new display type may be opened, certain messages must be implemented in the model's class, as described in section 5.3.

### 6.1. Creating the Display Type

This section explains creating the *customArrows* display type, its first recipe, and the specifications for this recipe. Again, Figure 6.3 shows the display type editor after *customArrows* has been added, using the *add* command in the display type list view. After adding the display type, we add the condition message for the first recipe, *isLeaf→()*, by executing the *add* command in the recipe list view, as shown in Figure 6.4. Next we add a new specification by

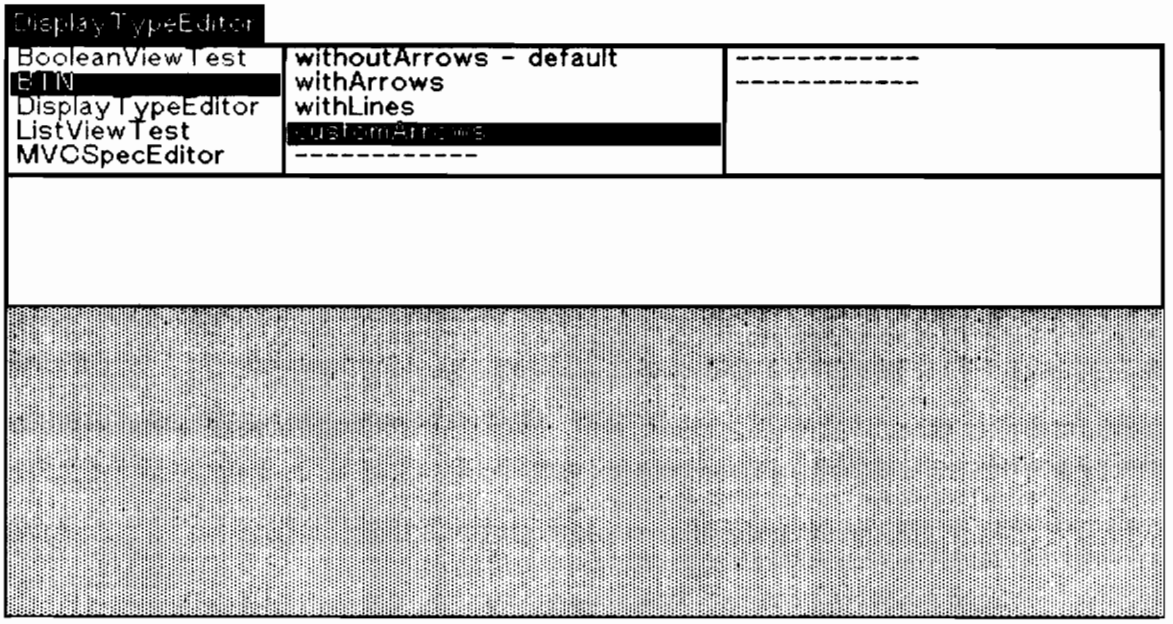


Figure 6.3 Adding the Display Type

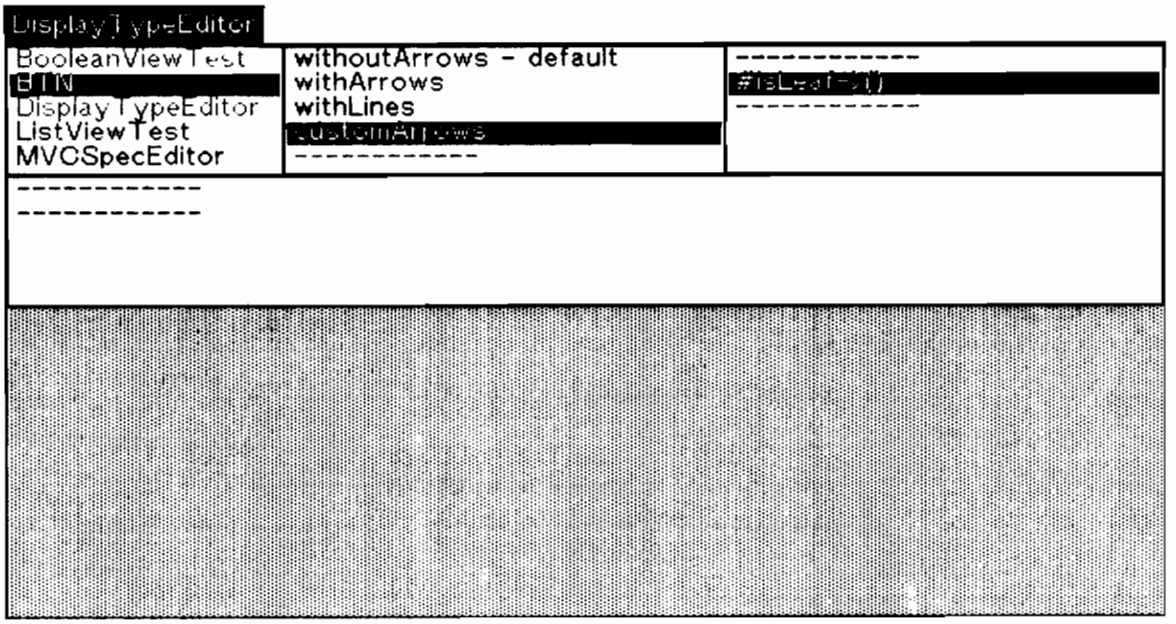


Figure 6.4 Adding the First Recipe

executing the *add* command in the specification list view. Figure 6.5 shows the display type

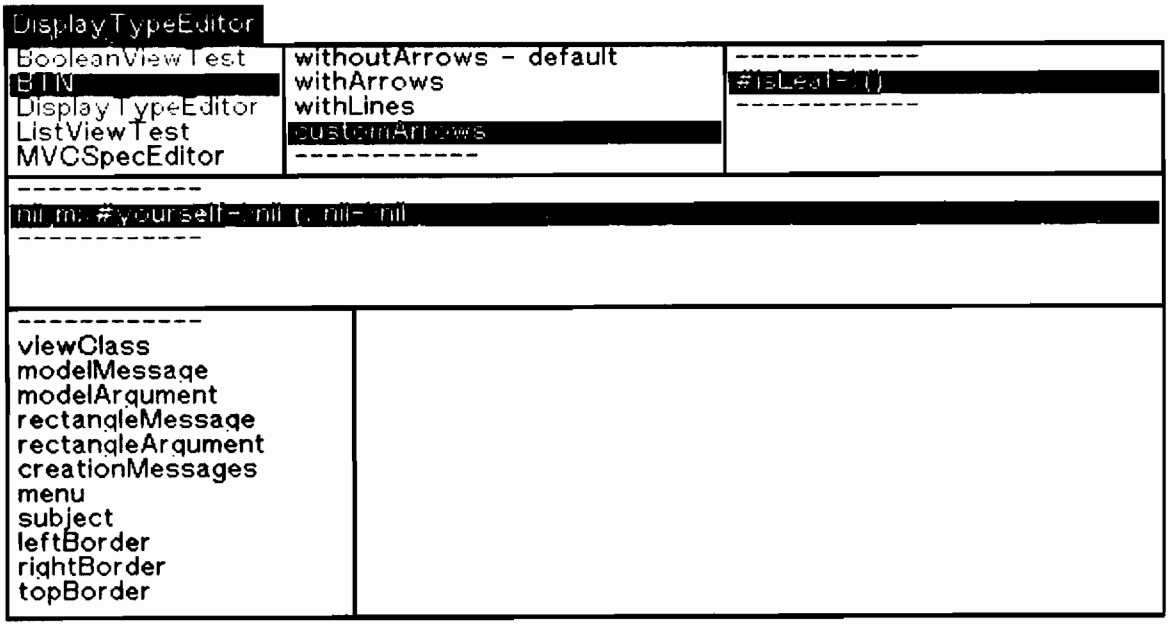


Figure 6.5 Adding a Specification

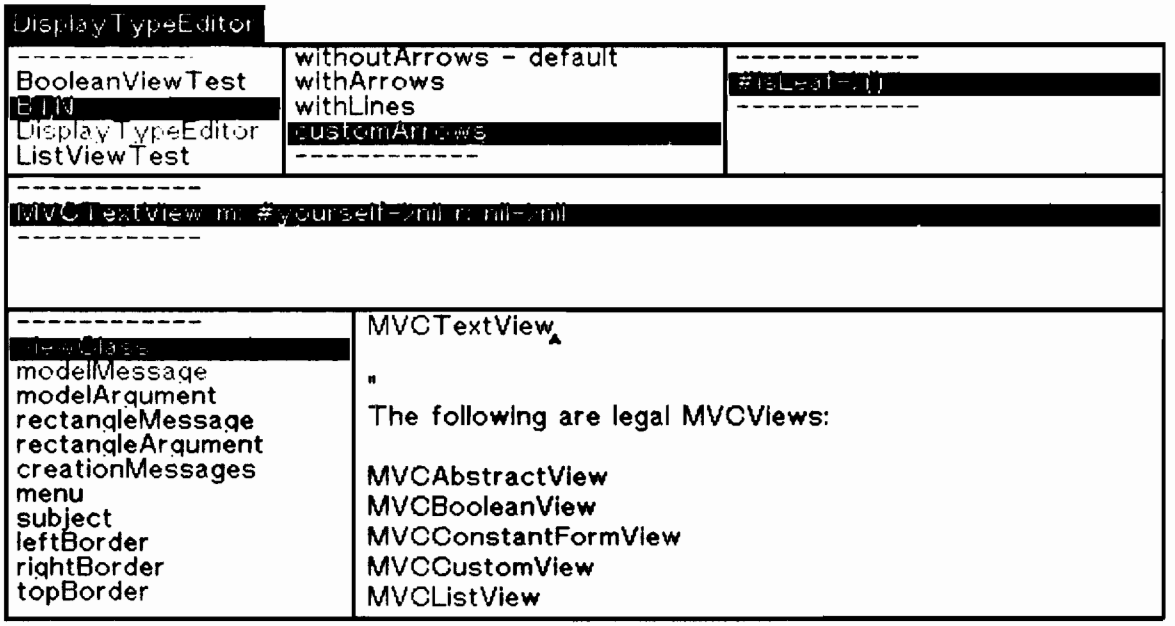


Figure 6.6 Editing *viewClass*

editor in edit mode immediately after this addition. Figure 6.5 shows that the newly added specification already has the message selector *yourself* as its model message, since most

specifications use *yourself* as their model message. Next we select the instance variable *viewClass* in view 5 of the display type editor. This action results in the value *nil* being displayed in view 6 along with a list of the legal MVC view classes. The value *nil* is then replaced by *MVCTextView*, as shown in Figure 6.6. The instance variable *viewClass* is edited first, because entering a value for *viewClass* causes the appropriate standard menu to be placed in the instance variable *menu*. All other parameters needed to specify *customArrows* can be entered in this way, or we can copy the *withArrows* display type, one of the existing display types for BTN, and modify its specifications.

During an ID, when the recipe *isLeaf→()* matches the model, one text view is added to the ID. The values for all the instance variables, including *viewClass*, in the specification created above are shown in Figure 6.7. In Figure 6.7, we see that the model message for this specification is *yourself* and *modelArgument* is *nil* ensuring that the model for the view described by this specification is an instance of class BTN. The instance variable *rectangleMessage* is *nil* and *rectangleArgument* contains the rectangle *(0@0 corner: 1@1)*. The rectangle in which this text view is added is (0@0 corner: 1@1). Thus the text view will occupy the entire space occupied by its superview. In Figure 6.1, the binary tree displayed has no children. The recipe *isLeaf→()* matches the model, so a text view was added in relative rectangle *0@0 corner: 1@1*. The area occupied by this view's superview, the seed abstract view for the ID, is the entire area occupied by the ID, so the text view also fills the entire ID. Two creation messages, the aspect message and the change message, are specified for the text view. Whenever the view receives *update: #textCopy*, it sends the message *textCopy* to its model. Whenever the user selects menu item *accept*, the model receives the message *accept: <aText>*, where *<aText>* is the text showing in the view. Figure 6.1 shows the menu available to users inside the text views in the ID shown. This menu contains the standard text editing commands and three additional commands: *add left*, *add right*, and *delete*. The standard text editing menu was supplied automatically when *viewClass* was given the value *MVCTextView*. To this standard menu were added

```
MVCTextView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0@0 corner: 1@1
    creationMessages: Dictionary (#aspect→#textCopy #change→#accept: )
    menu: ActionMenu
        labels: 'again
            undo
            copy
            cut
            paste
            do it
            print it
            accept
            cancel
            add left
            add right
            delete'
        lines: #(2 5 7 9 )
        selectors: #(#again #undo #copySelection #cut #paste #doIt #printIt
            #accept #cancel #addLeft #addRight #delete )
    subject: nil
    leftBorder: 1
    rightBorder: 1
    topBorder: 1
    bottomBorder: 1
    displayMessage: nil
    controllerClass: nil
```

Figure 6.7 Instance Variable Values for the First MVCSpecification

the menu labels *add left*, *add right*, and *delete* to give the menu shown in Figure 6.7. The list of specifications for the first recipe, *isLeaf→()*, contains just the single specification shown in Figure 6.7.

The second recipe, *isLeftChildNil→()*, has three specifications. The first is for a text view with the same functionality as the text view in the *isLeaf→()* recipe. The only difference between these two specifications is the size of the relative rectangle. When a binary tree node has a right child, we must leave room to display the right pointing arrow and the right child itself. In order to do this, the rectangle allotted to the text view must be smaller than (0@0 corner: 1@1). Instead, the specified rectangle is (0.25@0 corner: 0.75@0.2). The view displaying

the text *Top Node* in Figure 6.2 is displayed in relative rectangle (0.25@0 corner: 0.75@0.2). Observe that the left edge of this text view is one quarter of the way in from the left edge of the ID, and the right edge is one quarter of the way in from the right edge of the ID. Thus the view is centered in the x-range. In the y-range, it occupies the top fifth of the area allotted to the entire ID. Figure 6.8 shows outlines and relative rectangles for the three views added to an ID when recipe *isLeftChildNil→()* matches the model.

The second view we need to add for a tree node with no left child is the view containing the right-pointing arrow. The specification for this view designates a custom view as the *viewClass*. The model message, *yourself*, means the custom view will have a tree node as its model; the same node that is the model for the text view. The *rectangleArgument* stipulated is (0.5@0.2 corner: 0.75@0.3), so the origin of the custom view holding the arrow will be at the bottom center of the text view previously added, as shown in Figure 6.8. The text view extends from 0.25 to 0.75 in x-range, so 0.5 is at the center of this range; the y coordinate of the (lower

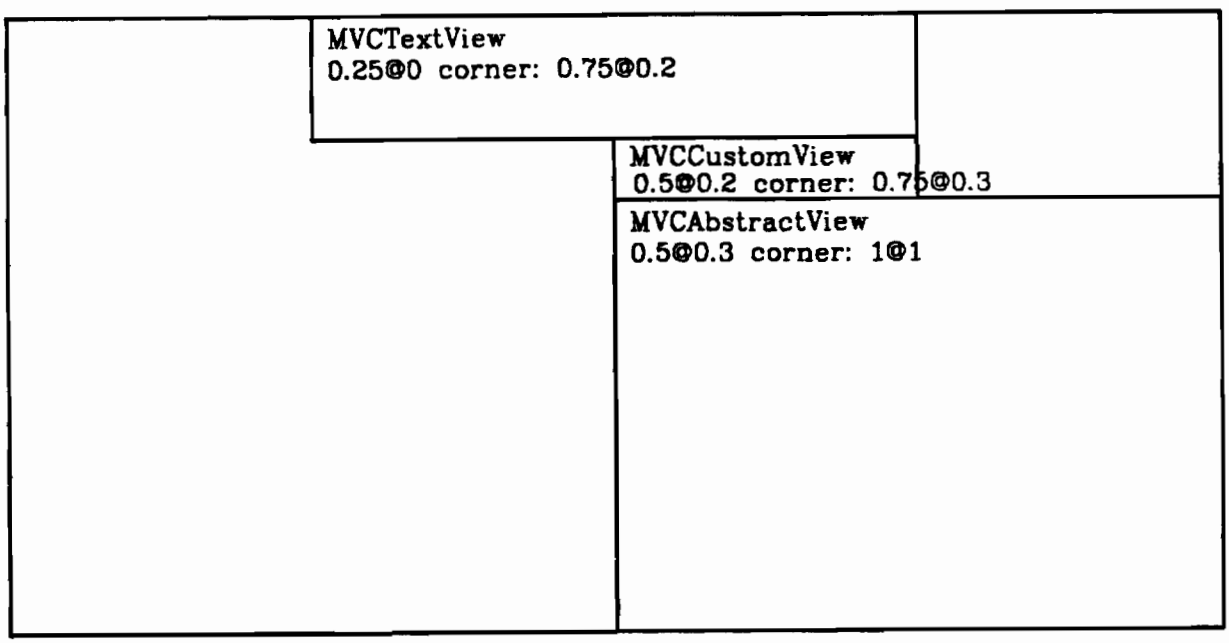


Figure 6.8 Use of Relative Rectangles

right) *corner* of the text view and the y coordinate of the origin of the custom view are both 0.2, so, in the y-range, the custom view starts where the text view leaves off. Figure 6.9 shows the specification for this custom view. From this figure, we see that this custom view has no creation messages. There is no menu, because no menu should appear in this view. All border widths are zero because we want the arrow to appear as if drawn in free space, connecting the views. Custom views have no inside color, so they take on the inside color of their superview light gray. Lastly, the message *rightArrow* is sent to the model every time the custom view is displayed. The method implementing this message draws a right-pointing arrow on the screen.

The last specification in recipe *isLeftChildNil→()* specifies an abstract view. This specification is shown in Figure 6.10. The model message for this abstract view is *rightChild*. During an ID, when this view is added, the parent view sends the message *rightChild* to its model, and the result becomes the model of the new abstract view. The relative rectangle for this abstract view, (0.5@0.3 corner: 1@1), extends from the bottom of the view displaying the right arrow to the bottom of the allotted area in the y-range, and from the center to the right edge of the allotted area in the x-range, as shown in Figure 6.8. Instance variables *creationMessages* and *menu* both have value *nil* in this specification. Abstract views are not required to have an aspect message. If they have none, their model message should return some subpart of

```
MVCCustomView
    modelMessage: #yourself
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0.5@0.2 corner: 0.75@0.3
    creationMessages: nil
    menu: nil
    subject: nil
    leftBorder: 0
    rightBorder: 0
    topBorder: 0
    bottomBorder: 0
    displayMessage: #rightArrow:
    controllerClass: nil
```

Figure 6.9 Specification for the Custom View

```
MVCAbstractView
    modelMessage: #rightChild
    modelArgument: nil
    rectangleMessage: nil
    rectangleArgument: 0.5@0.3 corner: 1@1
    creationMessages: nil
    menu: nil
    subject: 'customArrows'
    leftBorder: 0
    rightBorder: 0
    topBorder: 0
    bottomBorder: 0
    displayMessage: nil
    controllerClass: nil
```

Figure 6.10 Specification for the Abstract View

the model. In this case, we return the BTN instance variable *rightChild*. This view has no menu, and the borders are all zero width. Zero width borders are almost always used for abstract views, as they give the illusion that an abstract view is not present, but only its subviews. The value of *subject* for this abstract view is *customArrows*, the display type we are currently building. The display type entered in *subject* must be associated with the same class as the class of

(1) the model for the abstract view being specified, if this abstract view has no aspect message, or

(2) the object returned by the aspect message for the abstract view being specified, if this abstract view has an aspect message.

In this case, rule (1) applies. The abstract view being specified has no aspect message, as shown in Figure 6.10; its model is the right child of a tree node, another tree node; and the display type *customArrows* is associated with class BTN.

The next recipe is *isRightChildNil→()*. The views for this recipe are the same as those for *isLeftChildNil→()* except they put the rectangles for the custom and abstract views on the left instead of the right side. Finally, if none of the above recipes matches the model, *notNil→()*

will. In this case, we know the tree node for which we are adding views has two subtrees. Consequently, this recipe designates five views. Figure 6.11 shows the placement of these five views relative to one another. The text view is the same as in the last two recipes. Custom views for both the left and right arrows and abstract views for both the left and right subtrees are needed to complete the display.

### 6.2. How the Display Type is Used

The following example illustrates the process of matching a recipe and adding views to an ID based on the specifications associated with that recipe. Figure 6.12a shows an ID on a tree node, *Top*, that is a leaf. Figure 6.13a shows the ID after tree node *Right* has been added as the right child of *Top*. Figures 6.12b and 6.13b diagram all the views present in the ID at these two points in time. When the command in Figure 6.12a is executed, tree node *Top* executed the method *addRight*, which added tree node *Right* as the right child of *Top*, then executed *self changed*. When a view is created and given a model, this view is added to the end of the list of

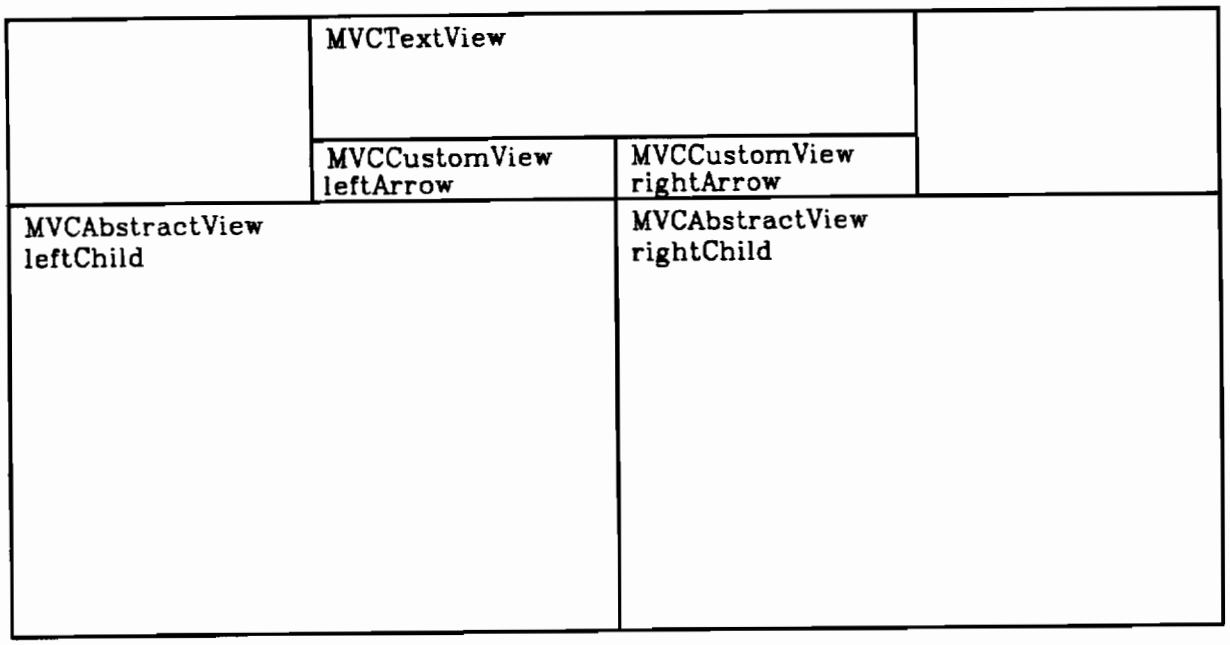


Figure 6.11 Views Specified by MVCRecipe *notNil→()*

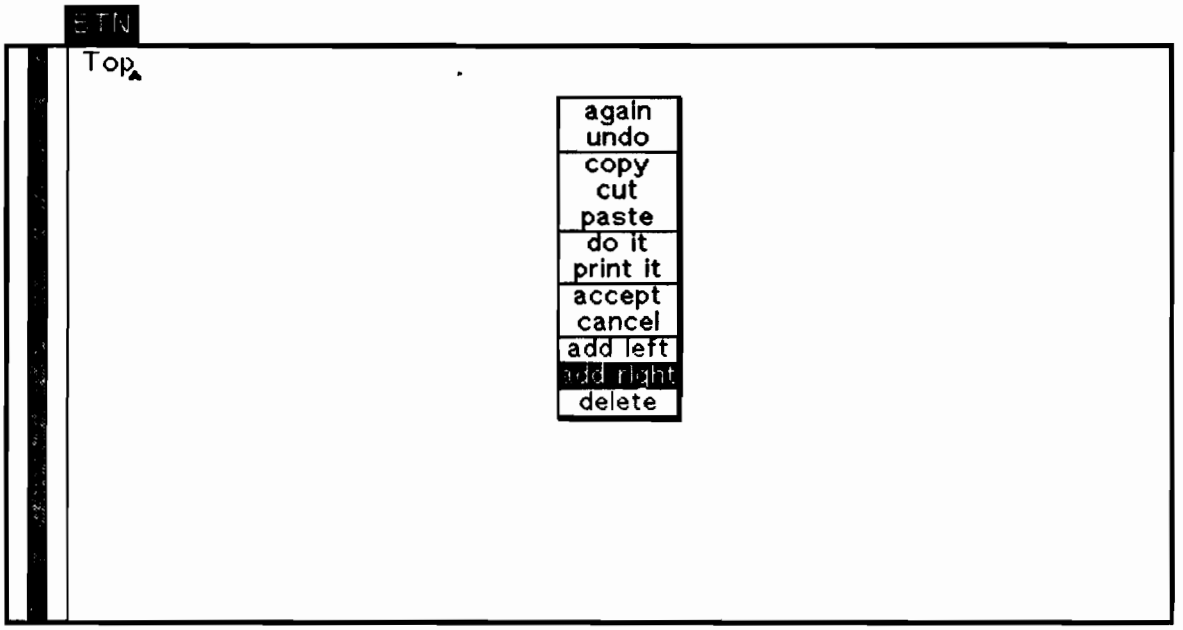


Figure 6.12a Before Adding Right Child

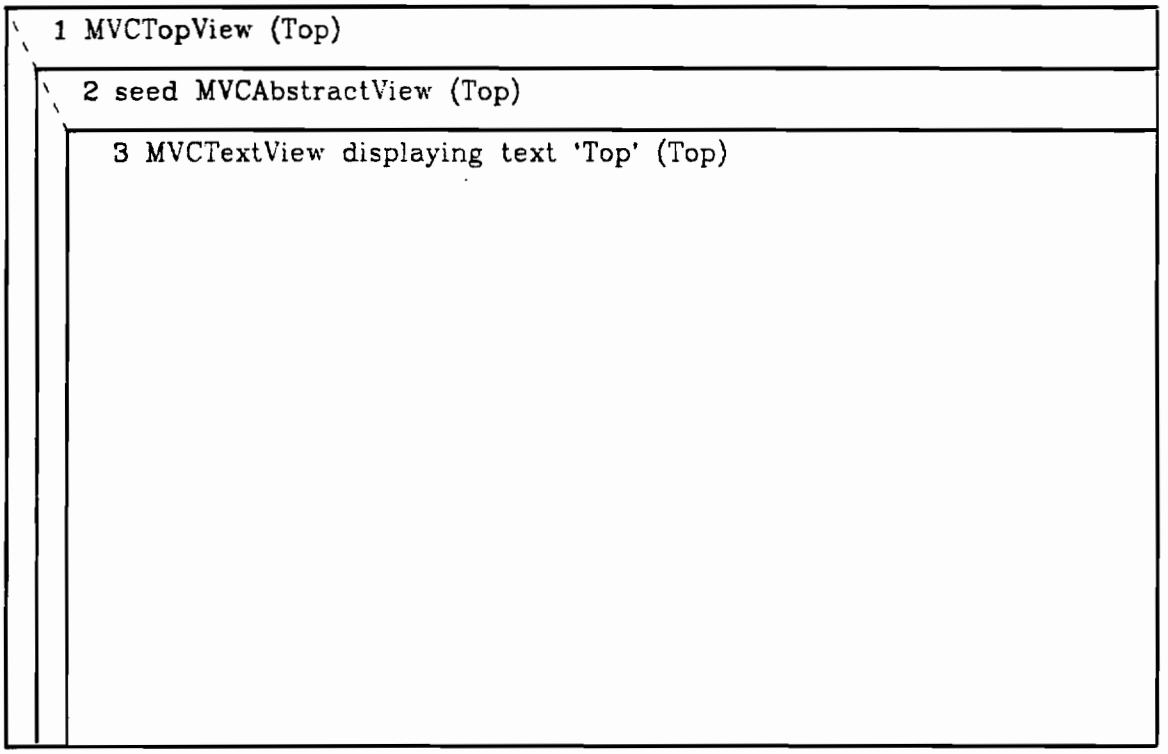


Figure 6.12b View Diagram for Figure 6.12a

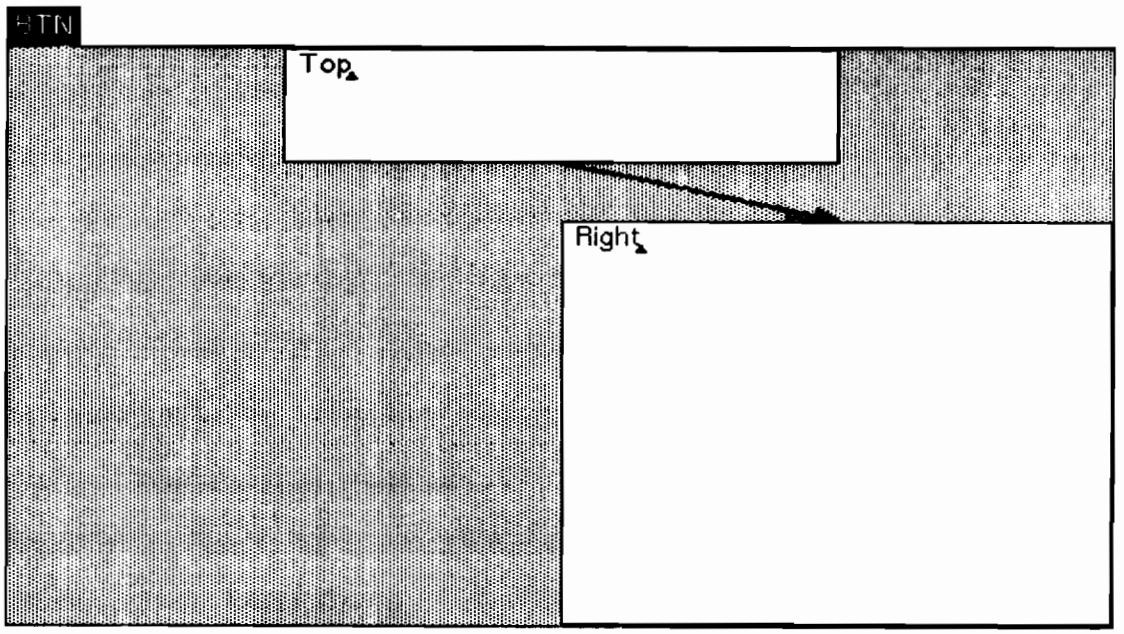


Figure 6.13a After Adding Right Child .

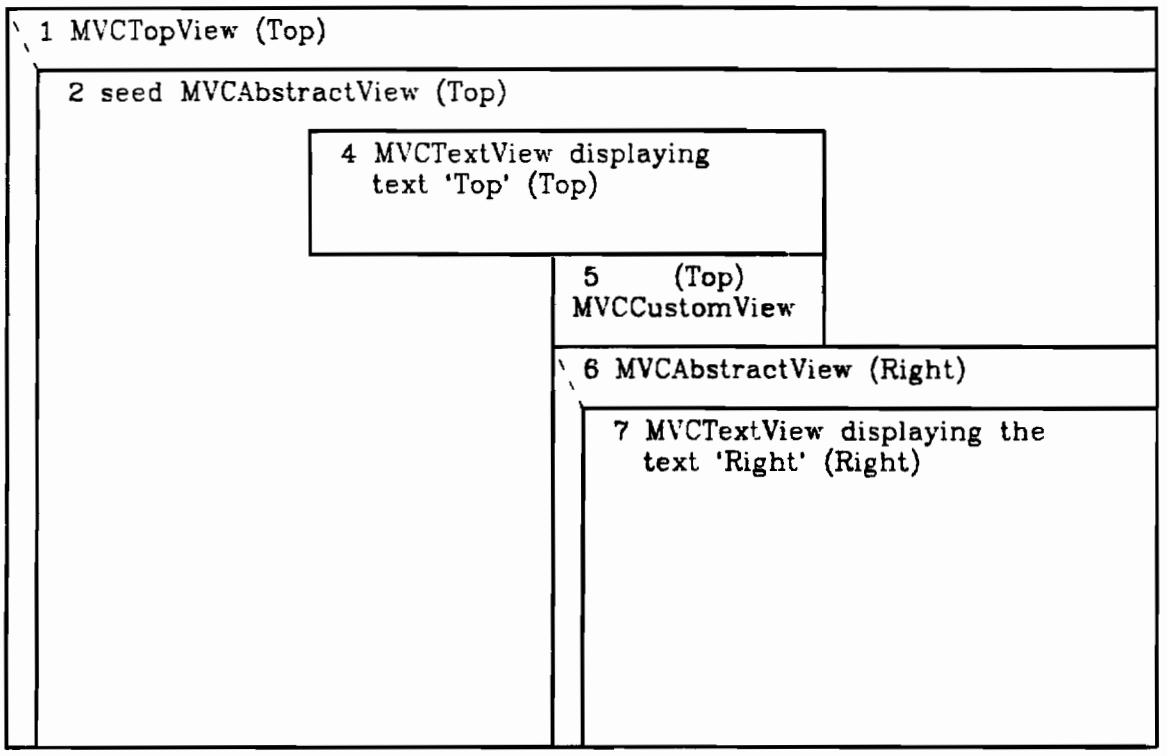


Figure 6.13b View Diagram for Figure 6.13a

views associated with this model. When the model executes *self changed*, the resulting *update:*

message is sent to each view in the model's dependency list in turn. In this example, the views are ordered in the dependency list as they are numbered in Figure 6.12b. The message *update:* with tree node tree node *Top* as argument was first sent to view 1 in Figure 6.12b. Top views ignore *update:*, so nothing happened. Then view 2 received the *update:* message. On receipt of the message *update:* <*anObject*>, an abstract view executes the method *addSubViews* if <*anObject*> is its model. This condition is met for view 2 in Figure 6.12b, so view 2 executes *addSubViews*. The method *addSubViews* takes the following action.

(1) Deletes existing subviews, such as view 3 in Figure 6.12b. Deleting a view removes it from the dependency list, so view 3 never receives the message *update:*.

(2) Display border and inside color. View 2 has a zero-width border, so displaying it does nothing. Abstract views have light gray inside color, so the entire space occupied by the ID is painted light gray.

(3) Determines the object to which to send the recipe condition messages. Since view 2 has no aspect message, the condition messages are sent to the model of view 2, the tree node *Top*.

(4) Gets the condition messages. View 2 holds the specification that describes it, from which it extracts the instance variable *subject*. The extracted value names the display type view 2 will use to add its subviews, i.e., *customArrows*. View 2 then asks *DisplayBuilder* to return the actual DisplayType object with this name. Upon receipt of this display type, view 2 picks out the collection of recipes for use in the next step.

(5) Sends the condition messages to the object determined in step (3). The first recipe for *customArrows* is *isLeaf→()*. View 2 sends the condition message *isLeaf* to tree node *Top*, which now has a right child, so it returns false. View 2 then sends the next condition message, *isLeftChildNil* to tree node *Top*. This condition is true of *Top*, so this recipe matches the model.

(6) Adds subviews according to the specifications associated with recipe *isLeftChildNil→()*. This recipe has three specifications, as discussed in Section 6.1.

(6a) As its first subview, view 2 creates and adds a text view in relative rectangle (0.25@0 corner: 0.75@0.2), as described in the first specification. This text view is view 4 in Figure 6.13b. Note that view 3 no longer exists. After adding view 4 as the first subview, view 2 sends the model message, *yourself*, to tree node *Top* to determine the model for view 4. Tree node *Top* is returned and installed as the model for view 4. Next view 4 is told to initialize itself. This initialization consists of: 1) recording the messages *textCopy* and *accept:* as the aspect and change messages, respectively, for view 4, 2) setting all four borders to a width of 1, and 3) storing the entire specification in the instance variable *mvcSpec*. Lastly, view 4 is told to display. It displays its border and inside color, white. It then sends its aspect message, *textCopy*, to its model, tree node *Top*, and the text 'Top' is returned. This text is then displayed in view 4, as shown in Figure 6.13a.

(6b) The second subview is a custom view. View 2 added it in relative rectangle (0.5@0.2 corner: 0.75@0.3). This custom view appears as view 5 in Figure 6.13b. After being added, view 5 is given tree node *Top* as model, in the same way as view 4. View 5 is then told to initialize itself. In this case, no change is effected in the display because view 5 has zero-width borders and custom views have no inside color. To complete the display, view 5 executes *model rightArrow: self*. This action sends tree node *Top* the display message, *rightArrow:*, with view 5 as the argument. The method *rightArrow:* then draws the arrow.

(6c) The third specification calls for an abstract view. View 2 creates a new abstract view and adds it in relative rectangle (0.5@0.3 corner: 1@1). This abstract view is view 6 in Figure 6.13b. To establish the model for view 6, view 2 sends the specified model message, *rightChild*, to the object determined in step (3), tree node

*Top*. This message returns the new right child, tree node *Right*, that is then installed as the model for view 6. View 6 is then instructed to initialize itself. At the end of the initialization, an abstract view executes *self addSubViews*. Therefore, view 6 performs the series of steps 1-6 with its model, tree node *Right*, and its subject, display type *customArrows*. Since tree node *Right* is a leaf, the recipe *isLeaf→()* matches it. View 7 in Figure 6.13b is added and displayed. View 7 occupies the same area as view 6. The display of view 7 completes the redisplay of tree node *Top*, and the result is shown in Figure 6.13a.

### 6.3. Code for BTN

Class BTN has four instance variables, *text*, *parent*, *leftChild*, and *rightChild*. In the display type above, several messages were specified. All of the messages not implemented in class Object must be implemented by BTN. To provide a cross section of the messages that must be implemented, the methods for messages *isLeaf*, *accept:*, *textCopy*, and *rightArrow:* are shown here.

The method for condition message *isLeaf* follows.

```
isLeaf
    "is this node a leaf"

    ↑leftChild isNil & rightChild isNil
```

This method is not too difficult; if instance variables *leftChild* and *rightChild* are both nil, the receiver is a leaf and returns true, otherwise false.

The creation messages *accept:* and *textCopy* are the change and aspect messages, respectively, for all text views in an ID using DisplayType *customArrows*.

```
accept: aText
    "Set the text of this BTN"
    text ←aText copy.
    self changed: #textCopy.
```

```
    ↑true

textCopy
    "Return a copy of the value of the instance variable 'text'."

    ↑text copy
```

The method for *accept:* takes one argument, the text in a text view when the *accept* command is executed, makes a copy of this text, and stores it as the label for the receiver, a tree node. Any text is acceptable, so the method always returns true. The method for *textCopy* simply returns a copy of the text currently stored in the receiver.

The message *rightArrow:* is the display message for the custom views displaying right pointing arrows. The method for *rightArrow:* uses an instance of class Pen to draw the arrow. Instances of class Pen exhibit behavior similar to that of the Logo turtle.

```
rightArrow: aView
    "Draw an arrow from the top right to the bottom left
    corner of aView's displayBox"

    | bic db pnt |
    bic ←Pen new.
    bic mask: Form black.
    bic defaultNib: 2.
    bic combinationRule: Form over.
    bic place: (db ←aView displayBox) topLeft.
    bic goto: db bottomRight.
    bic turn: 292.5.
    bic go: (pnt ←db extent // 5).
    bic place: db bottomRight.
    bic turn: 45.
    bic go: pnt
```

This method draws the line making up the shaft of the arrow then two small lines to create the tip. Note that the lower right corner of this view is at the top center of the abstract view holding the right child. This means the method should draw the arrow from the top left corner of the view that is its argument to the bottom right corner of this view. The message *displayBox*, sent to the argument *aView*, returns the rectangle on the screen that *aView* occupies. The

method then draws the arrow from the top left to the bottom right corner of this rectangle.

# 7. The Display Type Editor

The display type editor itself is an ID that was written using SIG. Two display types, *edit* and *edit or display*, specify the entire ID; they are associated with the classes DisplayTypeEditor and MVCSpecEditor, respectively. In the following discussion, *display type editor* refers to an ID and *DisplayTypeEditor* to the Smalltalk class. DisplayType editors have two parts, a part that relates a given specification to a recipe and thereby to a display type, and a part that provides editing capability on this specification. The first task is accomplished by the four uppermost views in a display type editor. As we have seen, the lists displayed in these views are structured in a dependence hierarchy. Each class has display types; each display type has recipes; and each recipe has specifications. The second task is achieved by the two lower views; the left view selects an instance variable and the right view allows data to be entered for that instance variable.

The display types *edit* and *edit or display*, are associated with classes DisplayTypeEditor and MVCSpecEditor rather than DisplayType and MVCSpecification because both types make use of list views. List views require the model to maintain view-specific information: the current selection. A problem arises if, for example, we have two display type editors open simultaneously, each editing different recipes for the same display type. Figure 7.1 shows this scenario. If both editors had the same DisplayType as model, it would be difficult for this DisplayType to maintain the current selections for both editors. Each display type editor has a different instance of class DisplayTypeEditor as model. A DisplayTypeEditor maintains the current selections for the four upper views of a display type editor. In the same way, the two lower views in a display type editor have an instance of MVCSpecEditor as model. A MVCSpecEditor points to a specification and keeps track of the instance variable currently selected.

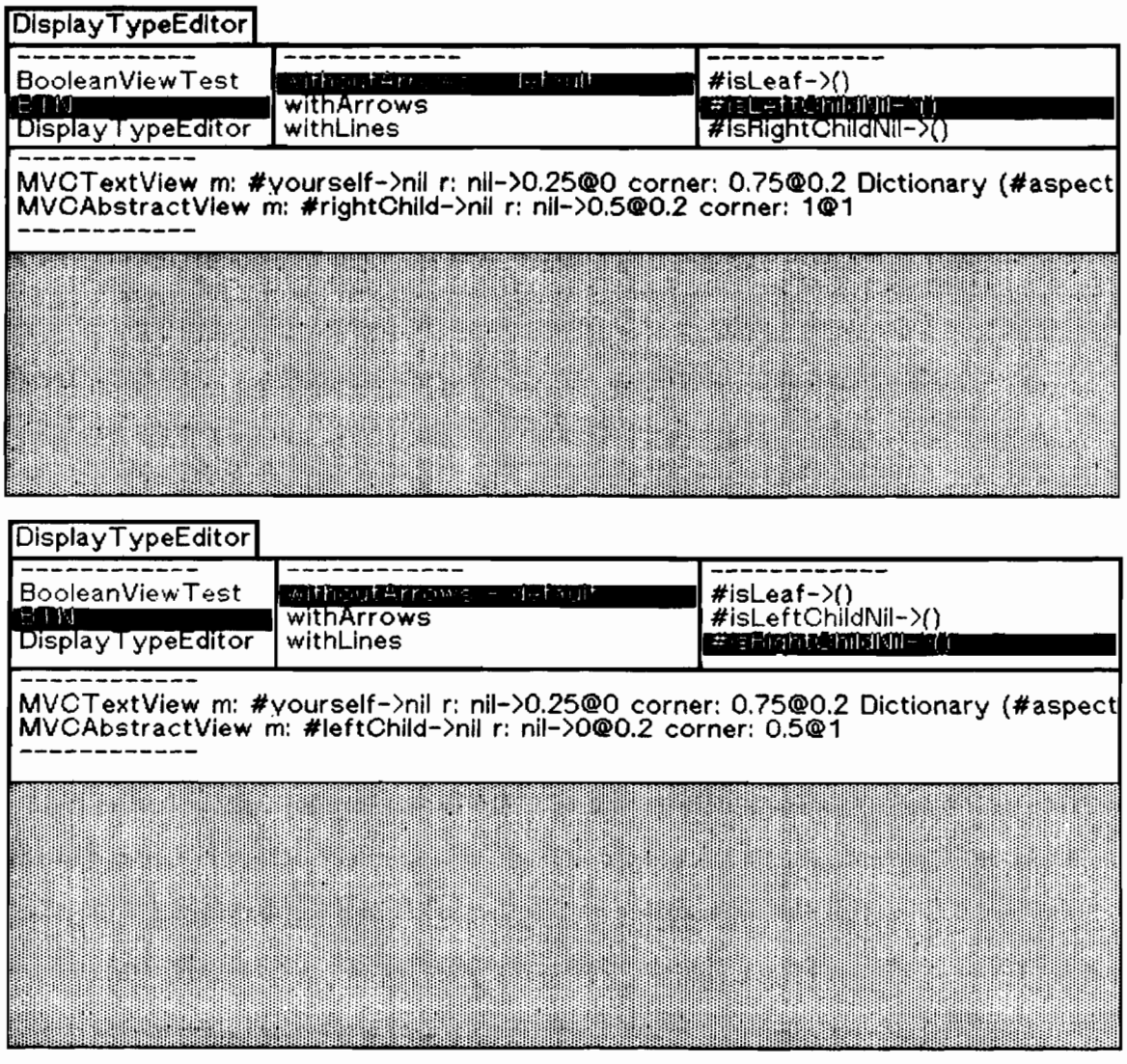


Figure 7.1 Two Display Type Editors on One Display Type

The display type *edit*, for class DisplayTypeEditor, has one recipe, *notNil→()*. This recipe has five specifications. The views designated by the first four specifications are the MVCList-Views that hold the list of classes, the display types for a class, the recipes for a display type, and the specifications for a recipe. To maintain the current selections in each of these list views, DisplayTypeEditors have four instance variables, *class*, *type*, *recipe*, and *specEditor*. The instance variable *specEditor* actually holds an MVCSpecEditor, not an MVCSpecification. This

anomaly is handled by the methods implementing the creation messages for the specification list view.

The fifth specification in the recipe *notNil→()* calls for an abstract view that has aspect message, *spec*; model message, *yourself*; and subject, *edit or display*. This abstract view holds a sub-ID, the visible views of which are views 5 and 6 of the display type editor, as shown in Figure 5.2. Since the model message is *yourself*, the model for this abstract view is the DisplayTypeEditor that is the model for the whole ID, but the subject, *edit or display*, is the name of a display type for the class MVCSpecEditor. To determine the model for views 5 and 6 of a display type editor, the abstract view that is the superview of views 5 and 6 sends its model the aspect message *spec*. This message returns the MVCSpecEditor found in the DisplayTypeEditor that is the abstract view's model. This MVCSpecEditor becomes the model for views 5 and 6.

The subject for the fifth specification in recipe *notNil→()* of display type *edit* is *edit or display*, which is the sole display type for class MVCSpecEditor. This display type has two recipes, *isDisplayFlagSet→()* and *notNil→()*. The first recipe has one specification that specifies a read-only view that displays instance variables and values for a specification. Figure 5.1b shows an example in which recipe *isDisplayFlagSet→()* matched. The second recipe represents the default case, edit mode, as shown in Figure 5.1a. This recipe contains two specifications, one specifying a list view for the list of instance variables and one specifying a text view for manipulating the selected instance variable.

Display type editors also appear as shown in Figure 7.1. Both of the display type editors in this figure have light gray space where the lower view(s) usually appear, because no specification has been selected yet. The instances of DisplayTypeEditor that are models for these editors have *nil* for *specEditor*. SIG assumes that nil should never be the model of a view, so abstract views do not even try to add any subviews if the model for those subviews would be nil. In each display type editor in Figure 7.1, the light gray appears where the lower views normally would, because no lower views were added.

# 8. Conclusion

SIG provides a structured mechanism for generating IDs in Smalltalk. Three innovations are provided in the SIG system.

(1) An ID is specified by display types that are outside of the ID itself.

(2) display types provide a way to add MVC triples to an ID conditionally.

(3) A mechanism is provided that interprets display types.

Together, these three features provide a uniform way to produce IDs.

## 8.1. Summary of the Features of SIG

Before SIG, the parameters of an ID were a part of the ID itself. Using only pluggable views, the yellow button menus that appear in any given view must be stored in the model, and each view that is part of an ID must know explicitly how to add subviews. That is, such views must know how many subviews to add, and the location and type of these subviews. Smalltalk inspectors provide a good example of this kind of ID. They all have two views, a list view and a text view, that always appear in the same positions in the ID. With SIG, information about the number and kind of views that are to appear in an ID is kept in a display type external to the views and controllers that make up the ID.

Views are added to an ID conditionally in SIG. This feature allows an ID to more accurately reflect the state of its model. Adding views conditionally allows the number, size, and type of views to change depending on the content of the model. This is particularly important for recursive models such as binary trees or Prolog literals. In these cases, it is impossible to specify a priori how many views will be needed. Even non-recursive IDs can benefit by adding views conditionally. A display type editor has two modes, edit and display, each requiring a different set of views. Display type editors can switch modes because views are added

conditionally.

Abstract views provide the mechanism by which display types are interpreted and views added to an ID. Whenever an abstract view is instructed to update, it uses a display type to add just those views that reflect the current state of the model. In addition, abstract views allow IDs to be composed. A display type can "call" another display type using its subject field. Thus, the specification of a large ID can be broken into subparts. The display type editor is such an ID. The four upper views are described by one display type, ***edit***. The subject of the last specification in the display type *edit* is ***edit or display***, the name of the display type that describes the one (or two) lower view(s) in a display type editor.

In SIG, the number of messages the model must implement is slightly increased over the number that must be implemented by the models of pluggable views. SIG models must implement the condition messages that are used to select recipes, although some of these messages would have to be implemented anyway, e.g., *isLeaf* for class BTN. Using SIG, the amount of code a user must write to implement an ID is reduced as no code must be written to implement views and controllers. Before creating SIG, I wrote an ID similar to the Prolog literal ID shown in the figures in Chapter 2. This ID required approximately twice as much code as the Prolog literal ID shown in Chapter 2. This decrease in code size is most significant for IDs with view hierarchies more than two levels deep. In addition, the tool set of primitive views combined with the mechanism for adding subviews gives a framework that cuts down the number of decisions to make when building IDs.

### 8.2. Problems with the MVC Paradigm

The Smalltalk MVC paradigm, while very versatile, is nonetheless quite difficult to use. There is no simple subset of things a novice user can do to produce an ID. At a minimum, classes for a model, view, and controller must be produced, thereby introducing the problem of understanding how these entities communicate, which is not under the user's control. In an ID

of any complexity, there are many views and controllers all tied to each other and their models in a hopeless entanglement. Views and controllers both store their corresponding controllers and views, and both store their models, but models know nothing about either their views or controllers. Models are related to their views, indirectly, through the dependency list. The only way for a model to affect the ID is to execute *self changed*, possibly with a parameter. This action either updates all the views on the model or, in the case of pluggable views, only those views whose aspect message matches the parameter. In either case, the model has no way of communicating to a view directly a request to update. Another problem is that control is passed from controller to controller via the *view* hierarchy. If a controller decides that it does not want control, the controller asks its view if any of the view's subviews want control. To execute this request, the view must ask each of its subviews in turn to ask their controllers if they want control.

Another problem with views and controllers is that controllers often update the display directly. Text controllers redraw the screen as text is being typed in, and they repaint the text caret inside a text view. List controllers reverse the display as the user moves the cursor over items in the list. Both of these actions are implemented in the controller for efficiency reasons. Response would simply be too slow a controller had to signal its view to make the update. This arrangement does work, but it contributes to the confusion and the whole mystical aura surrounding views and controllers.

Much of the problem lies in trying to integrate the viewing mechanism with the control mechanism. This problem has its roots in picture composition. If an image has structure, we want to treat different components of the structure in different ways. Usually, the ID designer wants to implement a *set* of behaviors. Each behavior is associated with some logical portion of the image. If the user is moving the cursor over an image on the screen, it is a difficult problem to determine to which logical portion of the image the cursor is pointing at any given time. This is not a problem as long as some screen space can be allotted to each logical portion of the

image. However, this space allocation can not always be accomplished. For example, Figure 8.1 shows a possible view scheme for an ID displaying a dictionary. This dictionary is a set of three associations. Each association has a key and a value. Together the three key-value pairs utilize all the space allotted to the ID. Each association has associated operations, e.g., text editing on the *value* of the association, which would appear whenever the user depressed the yellow button inside one of the views displaying an association. Suppose we want to add a new association to the dictionary. If we specify *add* as one of the menu messages to appear in a view displaying an association, an association receives the *add* message. Associations do not know if they are a part of some dictionary. The association in this example has no way to tell the dictionary of which it is a part to add a new association. One way around this problem is to allocate the dictionary itself some "real estate" in the ID, but this approach does not seem

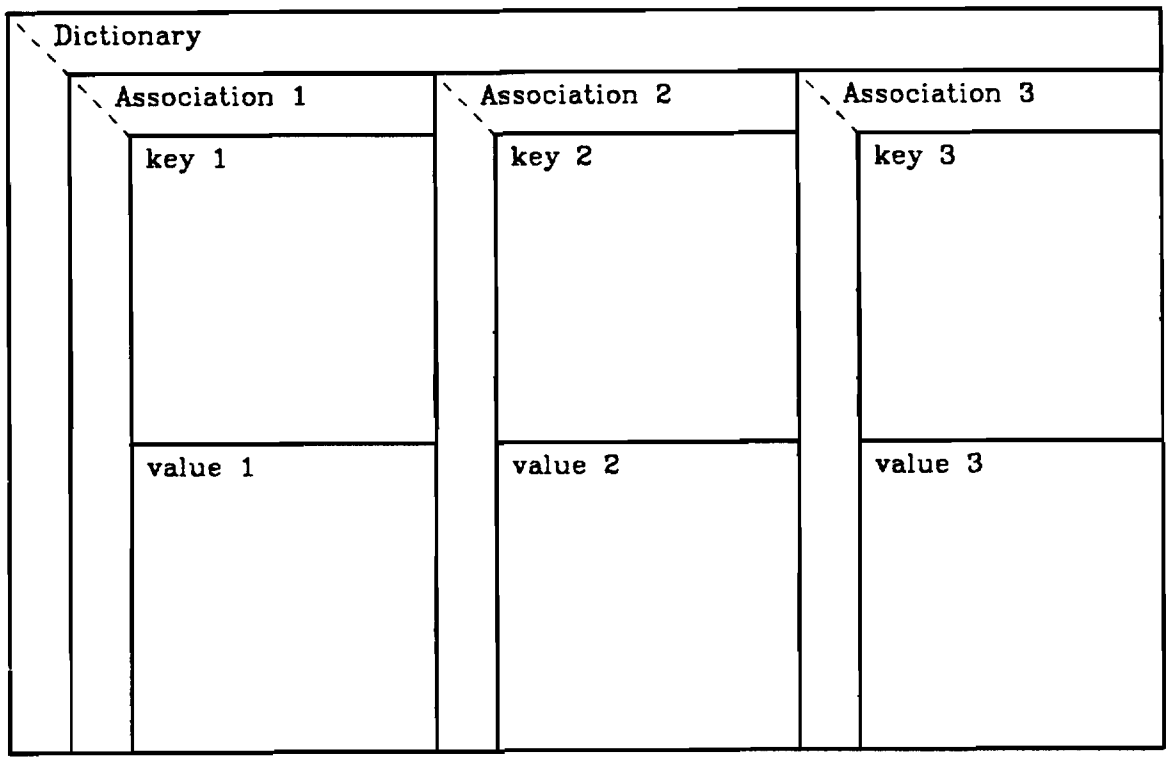


Figure 8.1 View Scheme for an ID on a Dictionary

right, as the set of associations *is* the dictionary. There is no information to be displayed in this designated "real estate"; it exists solely for control purposes. Another means of sending messages directly to the dictionary is to have a dedicated button. However, dedicating a mouse button is hardly practical in general as mice have a limited number of buttons while there may be arbitrarily many such buttons needed.

### 8.3. Limitations of SIG

Pluggable views ease the task of generating IDs by focusing attention on the model. Menu messages are sent to the model, and the model has indirect control over its views through the views' creation messages. SIG furthers this new focus. Display types provide a rudimentary mapping from a model to an ID on that model, albeit still using the existing MVC paradigm. SIG thus suffers from the "real estate" problem mentioned above. SIG extends the existing MVC paradigm by the addition of display types, but IDs specified by these display types are realized only by passing many messages between the views and models present in these IDs. The biggest problem with this scenario is that the user must understand, if not implement, all these messages. The model message and rectangle message, for example, mainly affect the mechanics of executing an ID, rather than what the ID actually displays. These messages can be confusing to the user.

### 8.4. Possible Solutions

The "real estate" problem can be addressed by specifying to which object a menu message should be sent. In the dictionary example above, the mapping might specify the following. If *add* or *delete* is selected, the corresponding message is sent to the dictionary. The other commands can be used to manipulate the text of a key or value. If the *accept* command is selected over a key, the original text for that key and the currently displayed text are sent to the dictionary. The dictionary then uses the original text to look up the key to be changed, removes it and readds the association with the new key.

In addition to such control specifications, viewing specifications would also be given. In this example, these viewing specifications might be something like the following.

(1) Keys are displayed above values.

(2) Each key-value combination extends from the top to the bottom of the display.

(3) Key-value combinations are displayed side by side.

In this viewing specification, there is no mention of associations or the dictionary, because they are not displayed as entities in themselves. Of course, in an actual specification, the alignment of displayed entities would have to be specified in more detail. Specifying which views go where in an ID might be done graphically instead of with relative rectangles as in SIG. The user could be allotted an area on the screen in which to draw rectangles in which data items would appear.

### 8.5. Reworking the MVC Paradigm

The problem of complexity in the MVC paradigm and the proliferation of messages specified in pluggable views and SIG DisplayTypes merit some additional work. Much of the confusion stems from the existence of three separate objects (the model, view, and controller) that are intimately related. Views and controllers are particularly interrelated, so much so that it is often quite difficult to determine which (the view or controller) is doing what when.

Future research in the area of making IDs in Smalltalk more understandable and easier to use effectively should concentrate on trying to find an alternative to the existing MVC paradigm. SIG is constrained to using the MVC paradigm. The MVC paradigm imposes the following restrictions.

(1) Each data structure component must have an associated view if it is to receive menu messages directly, hence the "real estate" problem.

(2) The user must have knowledge of the implementation of the views effecting an ID, because these views send specific messages to the data structure (model).

(3) When a change is made in a data structure, the data structure itself must announce that it has changed.

As a minimum requirement, any mechanism replacing the MVC paradigm has to overcome these three limitations. There should be a distinction between data structures (Smalltalk objects containing data in which the user is interested) and the viewing and editing of these data structures. How can these two things be related so as to produce effective IDs with a minimum of effort if not through the MVC paradigm? The idea of external specifications of mappings from data structures to IDs on these data structures, provided by display types in SIG, is a basis for a new approach to this problem. Rather than specifying which objects (views and controllers) to use to create an ID and the messages these objects understand, as is done in SIG, the external specifications should specify only the types of the components of the data structure, how these components should be positioned on the display relative to one another, and which components should receive which menu messages to effect changes in the data structure. SIG makes an effort at providing each of these things: typing, in the display type instance variable *viewClass*; relativity, in rectangle message; and control capability, in *menu*. However, the new specifications, unlike SIG's display types, must be as devoid as possible of any knowledge of the mechanism that interprets those specifications. This specification interpreter must monitor the user's data structure as well as the input devices. When a menu message input is received from the mouse, this interpreter determines, from the data structure specifications, to which component of the data structure to send this message. Since the interpreter is monitoring the data structure, if the data structure changes as a result of the input, the interpreter knows of this change and can update the display accordingly.

A mechanism that would return the value of any component of a data structure, given a specification for that component, could be implemented in class Object. For example, we may have a scheme for instances of class BTN. In addition to the display and menu information for BTNs, the scheme types instance variables. This scheme contains the fact that all BTNs have

four instance variables: *text*, *parent*, *leftChild*, and *rightChild* that have types Text | nil, BTN | nil, BTN | nil, and BTN | nil, respectively. During an ID, if the value of the right child of the left child of the root of a binary tree is needed, the specification interpreter can query the root by sending the message *get: 'leftChild.rightChild'*. The user does not have to implement a method for the message *get:*; it is implemented by class Object, and uses the typing information to carry out the query. Using a mechanism like this, the interpreter could extract the data it needed for any particular portion of a display from a data structure without the user being concerned with the way this extraction was accomplished.

Some problems would accompany the overthrow of the MVC paradigm. The most formidable of these problems is that all of the current Smalltalk IDs are written using the MVC paradigm. Although the MVC paradigm is pervasive throughout Smalltalk, it is not present at the lowest level of control and display manipulation. For example, instances of class ParagraphEditor, a kind of controller, contain the methods that update the display, monitor the keyboard, and implement yellow button menu items *again*, *undo*, *copy*, *cut*, *paste*, and *cancel*. ParagraphEditors do not use the MVC paradigm to accomplish these tasks; they access the display, mouse, and keyboard directly. It may be possible to use the code that accomplishes these actions more or less directly in a system that does not use the MVC paradigm.

# References


Baec81.
Ronald Baecker, "Sorting Out Sorting," 16mm color sound film, 25 minutes, SIGGRAPH 1981, Dallas, Texas (1981).

Brow84.
Marc H. Brown and Robert Sedgewick, "A System for Algorithm Animation," CS-84-01, Brown University, Dept. of Computer Science, Providence, Rhode Island (January, 1984).

Egge83.
Paul R. Eggert and Kam P. Chow, *Logic Programming Graphics and Infinite Terms,* Department of Computer Science, UCSB (June 1983).

Fisc84.
Gerhard Fischer and Matthias Schneider, "Knowledge-Based Communication Processes in Software Engineering," *Proceedings of the 7th International Conference on Software Engineering,* (March 1984).

Fole82.
James D. Foley and Andries Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley Publishing Company, Reading, Massachusetts (1982).

Furu82.
Richard Furuta, Jeffrey Scofield, and Alan Shaw, "Document Formatting Systems: Survey, Concepts, and Issues," *ACM Computing Surveys* **14 No. 3** pp. 417-472 (September 1982).

Gold84.
Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley Publishing Company (1984).

Hend82.
Peter Henderson, "Functional Geometry," pp. 179-187 in *ACM Conference on LISP and functional programming,* (August 1982).

Hero80.
C. F. Herot, R. T. Carling, M. Friedell, and D. Kramlich, "A Prototype Spatial Data Management System," *SIGGRAPH '80 Proceedings: ACM/SIGGRAPH Conference,* pp. 63-70 (1980).

Hero82.
Christopher F. Herot, Gretchen P. Brown, Richard T. Carling, Mark Friedell, David Kramlich, and Ronald M. Baecker, "An integrated Environment for Program Visualization," pp. 237-259 in *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development,* North-Holland Publishing Company (1982).

Kimu83.
Gary D. Kimura and Alan C. Shaw, "The Structure of Abstract Document Objects," Technical Report No. 83-09-02, Computer Science Dept., University of Washington, Seattle, WA (September 1983).

King84.
Roger King, "Sembase: A Semantic DBMS," *Proceedings of the First International*

*Workshop on Expert Database Systems*, pp. 151-171 ACM, (1984).

Knut79.
D. E. Knuth, *TEX and Metafont: New Directions in Typesetting*, Digital Press and the American Mathematical Society, Bedford, Mass., and Providence, R.I. (1979).

Kram83.
David Kramlich, Gretchen P. Brown, Richanr T. Carling, and Christopher F. Herot, "Program Visualization: Graphics Support for Software Development," pp. 143-149 in *IEEE 1983 Proceedings of the 20th Design Automation Conference*, (1983).

Lond84.
Ralph L. London and Robert A. Duisberg, *Animating Programs Using Smalltalk*, Computer Research Laboratory, Applied Research Laboratories, Tektronix, Inc., Beaverton, Oregon (1984).

Meyr82.
Norman Meyrowitz and Andries Van Dam, "Interactive Editing Systems: Parts I & II," *ACM Computing Surveys* **14 No. 3** pp. 321-415 (September 1982).

Myer83.
Brad A. Myers, "Incense: a System for Displaying Data Structures," *Computer Graphics* **17 no. 3** pp. 115-125 ACM, (July 1983). Also *Displaying Data Structures for Interactive Debugging*, Xerox PARC Reprot CSL-80-7,June 1980,xii+97 pp.

Pere83.
Fernando C. N. Pereira, *Can Drawing Be Liberated from the Von Neumann Style?*, Artificial Intelligence Center, SRI International (March 1983).

Reis83.
Steven P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," pp. 324-333 in *Proceedings - International Conference on Software Engineering*, IEEE (December 1983).

Wyk82.
Christopher J. Van Wyk, "A High-Level Language for Specifying Pictures," *ACM Transactions on Graphics* **1, No. 2** pp. 163-182 (April 1982).

# Appendix A

## Listing of the Code for Class BTN

```
Object subclass: #BTN
    instanceVariableNames: 'text parent leftChild rightChild '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SIG-Tests'
BTN comment:
'Binary Tree Node -- test class for Text & AbstractViews.
'


BTN methodsFor: 'text view adaptor'


accept: aText
    "Set the text of this BTN"
    text ←aText copy.
    self changed: #textCopy.
    ↑true


textCopy
    "Return a copy of the value of the instance variable 'text'."

    ↑text copy



BTN methodsFor: 'access'


leftChild
    ↑leftChild


leftChild: aBTN
    leftChild ←aBTN


parent
    ↑parent
```

```
parent: aBTN
    "aBTN might actually be nil, but that's OK"

    parent ←aBTN


replace: aChild with: aBTN
    "aChild should always be a BTN, aBTN may be nil.
    Replaces the child == aChild with aBTN."

    leftChild == aChild
    ifTrue: "its the left"
        [leftChild ←aBTN.
        ↑self].
    rightChild == aChild
    ifTrue: "its the right"
        [rightChild ←aBTN.
        ↑self].
    ↑self error: 'Couldn''t do the replace'


rightChild
    ↑rightChild


rightChild: aBTN
    rightChild ←aBTN



BTN methodsFor: 'testing'


isLeaf
    "is this node a leaf"

    ↑leftChild isNil & rightChild isNil


isLeftChildNil
    "is this node's left Child nil"

    ↑leftChild isNil


isRightChildNil
    "is this node's right Child nil"

    ↑rightChild isNil
```

*BTN methodsFor: 'tree functions'*

```
addLeft
    "add a new node where the receiver's leftChild is and ask whether
    the receiver's leftChild should be the left or right child of the new node."

    | newNode selection oldLC |
    self changeRequest
    ifFalse: "Someone doesn't want me to do this add"
        [↑self].
    (newNode ←BTN new) parent: self.
    leftChild isNil
    ifFalse:
        [selection ←0.
        [selection = 0]
        whileTrue: "Must select either left or right"
            [selection ←(PopUpMenu labels:
'make subtree left child of new node
 make subtree right child of new node ') startUp].
        selection = 1
        ifTrue: "subtree is left child"
            [newNode leftChild: leftChild]
        ifFalse:
            [newNode rightChild: leftChild].
        leftChild parent: newNode].

    leftChild ←newNode.
    self changed


addRight
    "add a new node where the receiver's rightChild is and ask whether
    the receiver's leftChild should be the left or right child of the new node."

    | newNode selection oldLC |
    self changeRequest
    ifFalse: "Someone doesn't want me to do this add"
        [↑self].
    (newNode ←BTN new) parent: self.
    rightChild isNil
    ifFalse:
        [selection ←0.
        [selection = 0]
        whileTrue: "Must select either left or right"
            [selection ←(PopUpMenu labels:
'make subtree left child of new node
 make subtree right child of new node ') startUp].
        selection = 1
        ifTrue: "subtree is left child"
```

```
            [newNode leftChild: rightChild]
        ifFalse:
            [newNode rightChild: rightChild].
        rightChild parent: newNode].

    rightChild ←newNode.
    self changed


delete
    "delete the receiver from the tree, and choose which subtree to make
    the new root.  Be sure to mark this node as deleted by giving it new text."

    parent isNil
    ifTrue: "Can't delete the root"
        [↑self error: 'Can''t delete the root'].
    (leftChild notNil and: [rightChild notNil])
    ifTrue: "Can't delete this node because it has subtrees"
        [↑self error: 'Can''t delete an interior node with left & right subtrees'].

    parent changeRequest
    ifFalse: "Someone isn't logically complete."
        [↑self].
    self changeRequest
    ifFalse: "Need to do this in case this node is at the top of an expanded view."
        [↑self].
    leftChild isNil
    ifTrue: "make the rightChild the new root"
        [rightChild isNil
        ifFalse: "non nil nodes must have their parent reassigned."
            [rightChild parent: parent].
        parent replace: self with: rightChild]

    ifFalse: "make the leftChild the new root"
        [leftChild parent: parent.
        parent replace: self with: leftChild].
    parent changed.

        "The next three lines totally disassociate this node from the tree."
    parent ←nil.
    leftChild ←nil.
    rightChild ←nil.

    "Mark this node as deleted (in italics)."
    text ←Text string: 'deleted' emphasis: 3.
    self changed "To warn the user this node is now deleted."
```

*BTN methodsFor: 'line drawing'*

```
leftArrow: aView
    "Draw an arrow from the top right to the bottom left
    corner of aView's displayBox"

    | bic db pnt |
    bic ←Pen new.
    bic mask: Form black.
    bic defaultNib: 2.
    bic combinationRule: Form over.
    bic place: (db ←aView displayBox) topRight.
    bic goto: db bottomLeft.
    bic turn: 22.5.
    bic go: (pnt ←db extent // 5).
    bic place: db bottomLeft.
    bic turn: 45.
    bic go: pnt


rightArrow: aView
    "Draw an arrow from the top right to the bottom left
    corner of aView's displayBox"

    | bic db pnt |
    bic ←Pen new.
    bic mask: Form black.
    bic defaultNib: 2.
    bic combinationRule: Form over.
    bic place: (db ←aView displayBox) topLeft.
    bic goto: db bottomRight.
    bic turn: 292.5.
    bic go: (pnt ←db extent // 5).
    bic place: db bottomRight.
    bic turn: 45.
    bic go: pnt
```

# Biographical Note

The author was born 7 April 1956, in Medford, Oregon. He attended the Medford public schools through grade 9. In 1971 he moved to Talent, Oregon and graduated from Phoenix High School in 1974. From 1974 through 1976, he attended Southern Oregon State College. In October 1976, he transferred to George Fox College, Newberg, Oregon where he received a Bachelor of Arts degree in Music Education and a Bachelor of Science degree in Math Education in June 1979.

From September 1979 to August 1982, the author managed the computer center at George Fox College. In September 1982 he began study at the Oregon Graduate Center and during his first year held a Clark Foundation Fellowship. He completed the requirements for the degree Master of Science in Computer Science in March 1983. He is leaving the Graduate Center to work as a systems software engineer with Intel Corporation, Hillsboro, Oregon.

The author has been married six and one half years to the former Christine Fitch.