The Filter Browser
Defining Interfaces Graphically

Raimund K. Ege[†], David Maier[†] and Alan Borning[‡]

Abstract

We describe a new approach to constructing user interfaces by declaratively specifying the relationships between objects via filters. A filter is a package of constraints dynamically enforced between a source object and a view object. For example, the relationship between an employee object in a database and a bitmap object represented on a display screen can be modelled as a filter. The object in the database can then be modified by manipulating the object on the screen, and changes made to the employee object by other programs are instantly reflected on the screen.

This paper describes a specification language for filters, an implementation of filters with the aid of a constraint-satisfaction system, and a graphical interface for designing filters. We illustrate the power and flexibility of the filter paradigm with an interface example and show that it stimulates and supports the re-use of existing components and gives a design methodology for constructing interfaces.

1 Introduction

This paper presents a new approach to building user interfaces in an object-oriented environment. In such an environment, all entities of interest are represented as objects, so all aspects of user interfaces are modelled as objects. In the Smalltalk model-view-controller (MVC) Paradigm [9], for example, the interface consists of model, view and controller objects. The model and view are basically two different representations of the same conceptual entity [6]. In Smalltalk's MVC paradigm, the model and view have procedural components that allow the controller to manage the interface correctly.

The Filter Paradigm: Our approach is to abandon procedural specification of user interfaces and relate the model (*source*) and *view* with a declarative interface specification. The idea is to use constraints to specify the conceptual relation between the *source* and *view* objects. For example, the relationship between an employee object and a bitmap object on a screen can be represented by constraints. The constraints state that the bitmap object always displays a certain rendering of the employee object. The constraints hide the procedurality of the interface. If the bitmap object on the screen is changed, then constraint satisfaction will ensure that the employee object is changed accordingly. If the employee object changes, then that change is reflected on the screen.

A *filter* is an object that describes and maintains these special constraints between objects in an interface. For example, consider an interface between two binary trees. The binary trees store an integer number at each node. The interface should be constructed in a way that one tree is the

Authors' addresses:
[†] Dept. of Comp. Science & Eng.
Oregon Graduate Center
Beaverton, OR 97006-1999
USA

[‡] Dept. of Comp. Science, FR-35
University of Washington
Seattle, WA 98195
USA

Research was supported by NSF grants IRI-8604923 and IRI-8604977.

reversal of the other tree, i.e., the interface can be viewed as the constraint that one tree is the mirror image of the other. This constraint can be represented as a filter between a binary tree as source and a binary tree as view object. Filters are constructed from subfilters on subparts of a source and view object. Figure 1 illustrates this tree reversal for two binary trees of height one constructed from three equality subfilters. The equality subfilters ensure that the integer numbers, stored in corresponding nodes, are kept equal. If a number is changed then the corresponding number on the other side of the filter is changed by the constraint-satisfaction system. If a subnode is added to a node on one side then a subnode is added by the constraint-satisfaction system on the other side of the filter and an equality subfilter is established between them. If a subnode is deleted from a node then the corresponding subnode is deleted from the other side of the filter and the equality subfilter is removed.

The definition of what types of the source and view objects are allowed for the filter and how the subfilters are connected to them is given by the *filter type*. Filter types specify how filters are built from atomic filters using set, iteration and condition constructors. Atomic filters are given by the implementation. The filter and object types are described by a filter specification language (*FiSpeL*) [8]. *FiSpeL* is currently a theoretical tool for composing filters; a compiler and optimizer for it are planned. The *Filter Browser* is a tool for constructing filters graphically. The *Filter Browser* lets the interface designer create filters by defining and manipulating filter types. Subfilters are added interactively by connecting them with the various constructors to the object types that are displayed in the browser. The *Filter Browser* also allows the designer to instantiate a filter from its filter type definition with sample objects to test the constructed interface.

Related Work: Our approach was guided by experience with the Smalltalk MVC paradigm [9]. Programming experience has shown that this paradigm is hard to follow. The Smalltalk Interaction Generator (SIG) tried to add a declarative interface on top of the MVC mechanism [12, 15]. One conclusion of SIG is that display procedures need type information about the objects they display and that Smalltalk does not provide this kind of typing. The Incense system [14] uses type information supplied by a compiler to display objects. The user can influence the display format but cannot update through this system. Editing can be used as an abstraction of user interaction [16] by separating the view from the source of data and defining a protocol for update. The Impulse-86 system [17] provides abstractions for objects as well as for interactions. Other approaches using algebraic techniques to specify user interfaces axiomatically [4] seem manageable only for small theoretical examples.

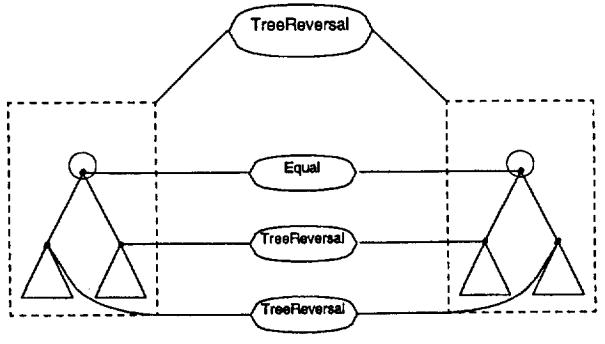

Figure 1: Tree Reversal Filter.

Constraints are used to specify relations and dependencies in Morgenstern's *active* database interface system [13]. Other systems use constraints as their major construct, such as ThingLab [1,2], which allows constraints to be expressed in a graphical manner. The Animus system [3,7] extends ThingLab with constraints that involve time. An early system that employed constraints to express graphical relations was Sketchpad [18]. The language Ideal [19], used in typesetting graphical pictures, is based on constraints and demonstrates their power and usefulness. Bertrand [11] is a term rewriting language that can specify constraint satisfaction systems. In its current implementation, however, it is not interactive and therefore not well suited for our problem. Constraints have also been used in the layout mechanism of a window management system [5].

Section 2 of this paper summarizes the filter specification language *FiSpeL*. Section 3 introduces the *Filter Browser* and elaborates an example session to generate an interface interactively. Section 4 gives details of how the objects, filters, constraints, and the *Filter Browser* are implemented in Smalltalk-80[1], and how the constraint satisfaction is performed by ThingLab [1]. The complete object and filter type definitions for the employee interface example from Sections 2 and 3 are listed in the appendix.

2 Objects and Filters

This section summarizes *FiSpeL*, a filter specification language for defining object and filter types. If we want to compose filters from subfilters, connecting objects of different kinds, it is necessary to type the objects to ensure that compositions of filters are well-defined. All entities in our filter paradigm are ultimately implemented by objects, so we put much effort in providing a comprehensive type system for the object model.

Object Types : The object type system supports the notions of aggregation and specialization. With aggregation we can build structured objects from components. Specialization allows us to refine existing objects via a hierarchy of object types and inheritance. We view object types as records. A record is a collection of typed fields. The fields have names called *addresses*. There are constant fields, which are constant for all instances of a type, and there are data fields, which are local to an instance of an object. Fields can be iterated by specifying an iteration factor; fields can be conditional by specifying a condition that must be true for the field to exist; and a field can specify its type

```
Object Type Person
      lastName → String
      firstName → String
      socSecNum → Integer
      address → String
end

Object Type Employee
      inherit from Person
      salary → Integer
      subNumber → Integer
      supervises[subNumber] → Employee
      supervisor → Employee
      jobDescription → String
end

Object Type DisplayEmployee
      icon → PersonIcon
      name → String
      subNumber → Integer
      subordinates[subNumber] → DisplayEmployee
end
```

Figure 2: Person and Employee object types.

[1] Smalltalk-80 is a trademark of Xerox Corporation.

recursively. In addition, an object type can inherit fields from other object types and can place constraints on all fields.

Figure 2 shows the two object types, `Person` and `Employee`. `Person` includes the fields that are common to a person. `Employee` is a specialization of `Person` and defines additional fields, such as `salary` and `jobDescription`. The field `supervises` is an iterated address, i.e., an employee may supervise a number of subordinates. The number of subordinates is specified by the field `subNumber`.

Object types are used in the filter type definition to describe source, view and variables that are needed to connect subfilters. A field of an object is accessed by traversing a path of field names. We distinguish direct and delayed access to an object via a path. In direct access, the structure of the object is traversed according to the path and the correct field is returned. In delayed access, we store the object identity together with the path to allow access on need at a later time. Delayed access is needed because it is possible that an object does not comply with the given path at the time when the path is defined (conditional fields), or the object at an end of a path may change before it is used.

Filter Types : The filter type system defines the structure of filters. Filters represent constraints between two objects. The filter type defines the types of the source and view objects it relates. The filter type also declares the subfilters that compose the filter. In addition, the filter type can define variables to be used as intermediate objects when subfilters are combined. A filter that is not further decomposed is called a *filter atom* and is directly supported by the implementation. For example, filter atoms are used for low-level input/output, data conversion, or error handling. A filter that has subfilters is called a *filter pack*. The subfilter constructors are: sequence, iteration and condition. The sequence constructor (`set of`) declares several subfilters of possibly different types; the iteration constructor (`iteration p times i`) declares a certain number of filters of the same type; the condition constructor (`condition`) declares a subfilter only if a given condition is true. It is possible to declare another instance of the same filter type as a subfilter of the one being defined, much like a recursive procedure call in a conventional programming language.

Figure 2 shows the object type definition for `DisplayEmployee` that contains displayable information for an employee. In order to define an interface that will display an object of type `Employee` we have to extract the displayable information. Figure 3 shows the filter type definition `EmployeeDisplay`. It decomposes the constraint into subconstraints. The subconstraints specify that the name, the number of subordinates and the subordinates are the same in the source object of type `Employee` and in the view object of type `DisplayEmployee`. The example uses the sequence and iteration filter constructors. The iteration names the same filter type for each of the subordinates recursively. The iteration depends on the value that is stored in `source.subNumber`. Whenever that value changes, the number of subfilters that are instantiated for the subordinates is changed. When including a subfilter, the filter type associates source and view paths with it. For example, the subfilter `StringEquality` has `source.firstName` as its source object. When this filter type is instantiated, the filter instance will check whether the address `firstName` in fact refers to an object of type `String`. It is at this point that the path to the field is traversed to retrieve it. The newly

```
Filter Type EmployeeDisplay ( source : Employee, view : DisplayEmployee )
      make set of
            StringEquality (source.firstName, view.name)
            IntegerEquality (source.subNumber, view.subNumber)
            iteration source.subNumber times i
                  EmployeeDisplay(source.supervises[i],view.subordinates[i])
end
```

Figure 3: EmployeeDisplay filter type.

defined filter type `EmployeeDisplay` can now be used as subfilter in other filter type definitions. The appendix contains the complete filter and object type definitions for a sample `EmployeeManipulation` interface.

3 The Filter Browser

The filter specification language is one way to define a user interface by defining filter types, but the goal here is to provide a graphical tool to build interfaces. Therefore, a tool similar to the Smalltalk [10], ThingLab [1] or Animus [7] browser, the *Filter Browser*, is used to define, manipulate and test filter types graphically. In defining filter types, we distinguish the external and internal parts of the definition. Externally, the filter type is identified by its name and the types of its source and view object. Internally the filter type specifies subfilters and variables. These details are encapsulated inside the filter type. A session with the filter browser has three different phases. In phase one, the name of the filter type and the type of source and view objects are given. In phase two, the variables and subfilters that participate in filter constructors, such as sequence, iteration and condition are specified. Phase one represents the external, phase two the internal definition. In phase three, a constructed filter type is instantiated. Figures 4, 5 and 6 show examples of the filter browser in each of the phases as we define the filter type `EmployeeManipulation`.

Phase one (Figure 4) : The filter browser is divided into several panes. In this phase only the upper left, lower left and lower right panes are used. The upper left pane shows a list of all filter types that are known to the system. The user can add a new filter type. The current subject of the filter type definition, i.e., the filter type that will be modified or newly defined, is emphasized. The filter browser displays two lists of all object types that are available for source and view types in the two panes below. The user can inspect all object types and select one for source and view. The object type also provides a sample instance (prototype) that can be used to instantiate the filter in phase three. The `EmployeeManipulation` filter type is defined on source objects of type `Company` and on view objects of type `FilterDevice` (these object types are emphasized).

Phase two (Figure 5) : The upper left pane displays a list of known filter types with the current selection, `EmployeeManipulation`, emphasized. To the right there are four panes to select the action that is to be performed on the current filter type. The user can *insert* subfilters, add

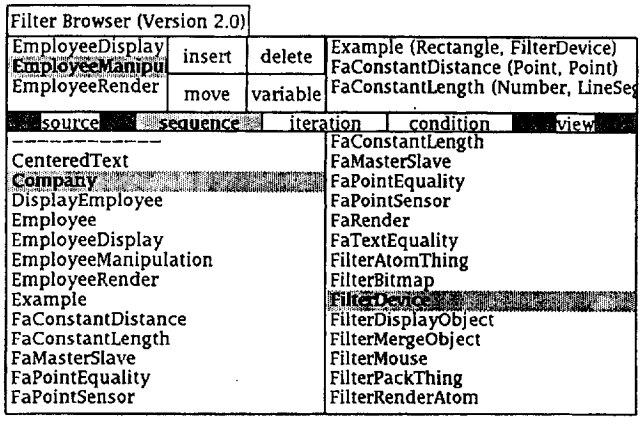

Figure 4: Filter Browser Phase One.

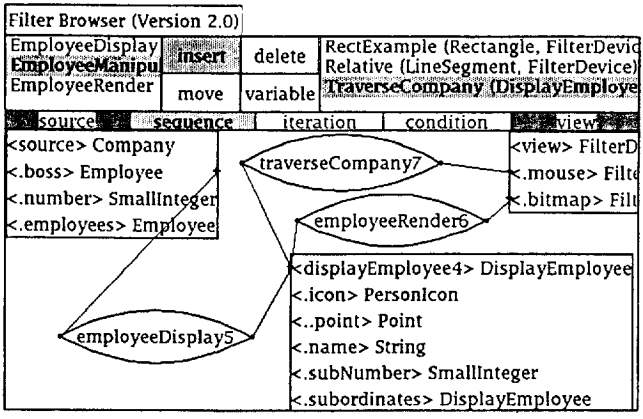

Figure 5: Filter Browser Phase Two.

variables, *move* or *delete* subfilters or variables in the picture pane below. The picture pane substitutes the two object lists from phase one. If the *insert* action is selected then the upper right pane shows a list of all filter types and the types of their source and view objects. This list also contains the currently defined filter type to allow the recursive specification of a filter type. The selected element in this list represents the object of the action that is performed, i.e., it names the filter type to be inserted as subfilter in the filter type. If the *variable* action is selected then the upper right pane shows a list of all available object types, from which the user may select. The type of filter constructor (sequence, iteration, condition) is selected with one of the three panes in the middle of the filter browser.

The picture pane is used to display the subfilters, variables and their connections. A variable is placed in the picture pane by selecting the *variable* action. The variable appears as a box that shows its name and type and the paths to its fields with their types. The user selects a location and places the variable. The variable is then inserted into the current filter type definition. A subfilter is placed in the picture pane by selecting the *insert* action and a subfilter from the upper right pane. The added subfilter is then connected (linked) to paths in either the source, view or variable objects by pointing at their location on the screen. For iteration or condition constructors, an iteration or condition object has to be selected by pointing to a path that represents the iteration factor or the condition. After the subfilter is placed in the picture pane it is inserted into the current filter type definition.

Figure 5 shows the filter browser as the user inserts the `TraverseCompany` subfilter into the `EmployeeManipulation` filter type. A variable of type `DisplayEmployee` has already been defined and connected to an `EmployeeRender` and `EmployeeDisplay` (see Section 2) subfilter. The `EmployeeRender` filter renders an object of type `DisplayEmployee` on the display bitmap. The `TraverseCompany` filter allows the selection of an employee within a company using a pop-up menu. These filters are already defined. After the *insert* action has been selected and the user moves the mouse cursor into the picture pane a lozenge for the `TraverseCompany` filter appears. The source link is connected to the `displayEmployee4` variable and the view link is connected to the `bitmap` address of the view object.

The picture pane represents the network of subfilters. The absolute position of the subfilters in the picture pane is not important, but it will be stored to redraw the subfilter network in the same way as it was defined. An important issue in connecting subfilters is typing. The external definition of a subfilter specifies the object type for its source and view object. Thus, when a source or view link of a subfilter is connected to addresses of source, view or variable objects of the currently defined filter

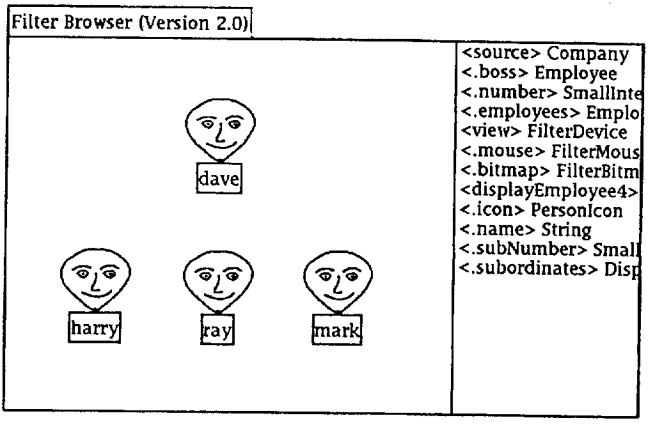

Figure 6: Filter Browser Phase Three.

type, it is necessary to check the corresponding types. These source and view links can be connected to object types that are of the same type as, or are subtypes of, their specified object types. For example, the source of the `TraverseCompany` filter has to be of type `DisplayEmployee` or one of its subtypes.

Phase three (Figure 6) : Instantiating a filter type means creating a Smalltalk object representing a filter. When instantiating, source, view and variable object instances of the correct object types have to be supplied. This instantiation operation is accessible from a pop-up menu in the upper left pane of the browser, where the filter type has been selected. The right pane of the filter browser simulates the display bitmap for the filters and all input is controlled by the filter browser, so it is possible to switch back to the previous phases. The right pane of the filter browser shows the participating instantiated objects. They can be selected, inspected and changed. Any change to participating objects will immediately change the display in the left pane. Figure 6 shows the instantiated `EmployeeManipulation` filter type for a prototype `Company`. The `FilterDevice` is simulated by the filter browser. The pane on the left displays the company's employees. We can traverse the tree of employees and make changes to the employees as they are stored for the company. The appendix lists the complete filter and object type definitions.

In phase three the user can test the filter type by observing its behavior and changing values of variables. The filter type can be changed incrementally. The user can switch back and forth between phases two and three, adding and deleting subfilters and variables or changing values of participating objects. All filter types are constructed from existing filters in a bottom-up fashion. The filter browser lists all existing filters together with their source and view type information. Filter types can only be constructed if the subfilters and variables are connected correctly and the filter can be instantiated while it is being developed to test its behavior. Thus the filter paradigm stimulates and supports the re-use of existing components and gives a design methodology for constructing interfaces.

4 Implementation

The *Filter Browser* is implemented on a Tektronix 4400 Series machine in Smalltalk-80[2]. ThingLab [1], an extension to Smalltalk, is used to do the constraint satisfaction. Filter and object

[2] Smalltalk-80 is a trademark of Xerox Corporation.

types in *FiSpeL* are represented as classes in Smalltalk. ThingLab extends the Smalltalk class definition with constraints, types for instance variables and dynamic access.

Object Types : Object types are modelled as Smalltalk classes. The fields of an object are instance variables, where the instance variable name is the address of the field. Iterated fields are represented as a set of fields, while conditional fields hold the value "nil" if the condition is not true. Fields can be inherited from superclasses. Field access is done through an access path that stores the field names that have to be traversed. Smalltalk does not keep information on the type of instance variables, so ThingLab augments the class definition to hold the type (reference to another object type class) for each instance variable[3]. Constraints that are defined within the object type are also stored in the class definition. An instance of an object type refers back to its class definition, so the constraint-satisfaction mechanism can retrieve the defined constraints.

Filter Types : Filter types are represented as subclasses of the object type classes. Fields are defined for the source and view objects, as well as for the variables. The subfilters are held in instance variables that can be conditional or iterated. The subfilters have source and view object information associated with them, i.e., what fields are used as source and view object for the subfilter. This information is modelled as equality constraints[4] from the field within the source, view or variable object to the appropriate source or view object of the subfilter. For example, consider the `EmployeeDisplay` filter type in Figure 3. The `StringEquality` subfilter is held by a sequence filter constructor; its source is associated with the `source.firstName` field of the filter type. This association is modelled by an equality constraint defined for the `EmployeeDisplay` filter type class.

ThingLab : As mentioned earlier, the filter browser is implemented as an extension to ThingLab. Path access using addresses (field names) and the constraint satisfaction is handled by ThingLab. The field description within ThingLab was augmented to incorporate iterated and conditional fields. Filter atoms are directly implemented as ThingLab constraints. ThingLab's prototypes of things are reduced in their importance in that they are used to hold the graphical information used to display the filter network in phase two and to provide sample values for the instantiation in phase three of the filter browser, but they no longer are used to infer the type of an instance variable.

Not all objects will be defined within ThingLab. Objects outside ThingLab are the display bitmap, the keyboard, specialized input devices (mice), or existing complex objects in an application. These object can be incorporated by either providing a special object in ThingLab that holds the outside object and controls all accesses to it (object holder), or by providing special filters that link existing objects within ThingLab to those external objects (implementation filter atoms). Graphical primitives, such as line rendering or input sensoring [8], are examples for filter atoms to incorporate I/O-objects.

5 Summary and Conclusion

The filter paradigm represents a new approach to interfaces in an object-oriented environment. Constraints are used as the basic building block for interfaces. Constructors are provided to allow building of structured interfaces in a declarative way. The feasibility of the filter paradigm is shown by providing a working interface generation tool. In addition, the object type system represents another approach to bring typing into Smalltalk.

Acknowledgements

This research has been funded by the National Science Foundation under Grants No. IRI-8604923 and IRI-8604977. Thanks also to Casey S. Bahr for many discussions that helped to refine the ideas for the implementation of the filter browser.

[3] Thus, at present, our typing mechanism does not allow an instance of a subtype to be stored in a typed instance variable. This is clearly an undesirable limitation, which we plan to remove. Removing it is not trivial, however, since depending on how the instance variable is used, we may need to ensure that the constraints on a subtype are not more restrictive than those on the specified type.

[4] Note that we do not distinguish between unification and equality constraint.

Bibliography

[1] Borning, Alan, *ThingLab - A Constraint-Oriented Simulation Laboratory*, PhD Thesis, Stanford University, 1979.

[2] Borning, Alan, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Trans. Prog. Lang. and Systems 3*, 4 (October 1981), 353-387.

[3] Borning, Alan and Robert A. Duisberg, Constraint-Based Tools for Building User Interfaces, *accepted for ACM Transactions on Graphics 5*, 4 (October 1986), .

[4] Chi, Uli, Formal Specification of User Interfaces: a Comparison and Evaluation of four axiomatic Methods, *IEEE Trans. on Software Eng. SE-11:8* (August 1985), 671-685.

[5] Cohen, Ellis S., Edward T. Smith and Lee A. Iverson, Constraint-Based Tiled Windows, *IEEE Computer Graphics and Applications*, May 1986.

[6] Deutsch, L. Peter, Panel: User Interface Frameworks, *OOPSLA'86 Conf. Proc.*, Portland, OR, September 1986.

[7] Duisberg, Robert A., *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System*, Ph.D. Thesis, Department of Computer Science, University of Washington, 1986.

[8] Ege, Raimund K., The Filter - A Paradigm for Interfaces, Technical Report No. CSE-86-011, Oregon Graduate Center, Beaverton, OR, September 1986.

[9] Goldberg, Adele and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, Mass., 1983.

[10] Goldberg, Adele, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, 1984.

[11] Leler, Wm, *Specification and Generation of Constraint Satisfaction Systems using Augmented Term Rewriting*, PhD Thesis, The University of North Carolina at Chapel Hill, 1986.

[12] Maier, David, Peter Nordquist and Mark Grossman, Displaying Database Objects, *Proc. First Int. Conf. on Expert Database Systems*, Charleston, South Carolina, April 1986.

[13] Morgenstern, M., Active Databases as a Paradigm for Enhanced Computing Environments, *Proc. 9th Int. Conf. on Very Large Data Bases*, Florence, Italy, October 1983.

[14] Myers, B., INCENSE: A System for Displaying Data Structures, *Computer Graphics 17(3)* (July 1983), 115-125.

[15] Nordquist, Peter, Interactive Display Generation in Smalltalk, Master's thesis, Technical Report CS/E 85-009, Oregon Graduate Center, March 1985.

[16] Scofield, J., *Editing as a Paradigm for User Interaction*, PhD thesis, University of Washington, Computer Science Department Technical Report 85-08-10, August 1985.

[17] Smith, Reid G., Rick Dinitz and Paul Bart, Impulse-86: A Substrate for Object-Oriented Interface Design, *OOPSLA'86 Conf. Proc.*, Portland, OR, September 1986.

[18] Sutherland, I., *Sketchpad: A Man-Machine Graphical Communication System*, PhD Thesis, MIT, 1963.

[19] Van Wyk, C., IDEAL User's Manual, Computing Science Technical Report No. 103, Bell Laboratories, Murray Hill, 1981.

Appendix

Section 2 and 3 described object and filter types as examples. The object and filter types given here incorporate those into an interface example that allows the manipulation of a sample employee database. Object or filter types from Figure 1 and 2 are not repeated here. The types `Integer`, `InputSensor`, `Interval`, `String`, `FormRender`, `StringRender`, `IntegerEquality`, `IntegerMultiply`, `IntegerDivide`, `BitStreamEquality`, `StringConversion`, `StringConcat`, `StringSensor` and `PopUpMenu` are atomic and given by the implementation.

```
Object Type Form
      width → Integer
      height → Integer
      count → Integer
      bits[count] → Bit
      constraint IntegerMultiply((width,height), count)
end

Object Type Company
      boss → Employee
      number → Integer
      employees[number] → Employee
end

Object Type PersonIcon
      inherit from Form
      constraint IntegerEquality (width, 30)
      constraint IntegerEquality (height, 45)
      constraint BitStreamEquality ((bits,(0@0), '0000100001000010000l...')
end

Object Type Device
      display → Form
      sensor → InputSensor
end

Filter Type ExtractHorizontal (source: (Form, Interval), view: Form)
      var
            ratio → Integer
            from → Point
      make set of
            IntegerEquality(source.width, view.width)
            IntegerMinus((source.second.high, source.second.low), ratio)
            IntegerMultiply((source.first.height,ratio), view.height)
            IntegerEquality(0, from.x)
            IntegerMultiply((source.first.height,source.second.low), from.y)
            BitStreamEquality((source.first.bits,from), view.bits)
end

Filter Type ExtractVertical (source: (Form, Interval), view: Form)
      var
            ratio → Integer
            from → Point
      make set of
            IntegerEquality(source.height, view.height)
            IntegerMinus((source.second.high, source.second.low), ratio)
            IntegerMultiply((source.first.width,ratio), view.width)
            IntegerEquality(0, from.y)
            IntegerMultiply((source.first.width,source.second.low), from.x)
            BitStreamEquality((source.first.bits,from), view.bits)
end

Filter Type EmployeeRender (source: DisplayEmployee, view: Form)
      var
            topForm, middleForm, bottomForm → Form
            verticalForms [source.subNumber] → Form
            formInterval [source.subNumber] → Interval
      make set of
            ExtractHorizontal ((view,(0,0.25)), topForm)
            ExtractHorizontal ((view,(0.25,0.3)), middleForm)
            ExtractHorizontal ((view,(0.3,1)), bottomForm)
            FormRender (source.icon, topForm)
            StringRender (source.name, middleForm)
            iteration source.subNumber times i
                  IntegerDivide ((i-1, source.subNumber), formInterval[i].low)
                  IntegerDivide ((i, source.subNumber), formInterval[i].high)
                  ExtractVertical ((bottomForm, formInterval[i])), verticalForms[i])
                  EmployeeRender (source.subordinates[i], verticalForms[i])
end

Filter Type EmployeeManipulation ( source : Company, view : Device )
      var
            bossDisplay → DisplayEmployee
      make set of
            EmployeeDisplay (source.boss, bossDisplay)
            EmployeeRender (bossDisplay, view.display)
            TraverseCompany (bossDisplay, view.sensor)
end
```

```
Filter Type TraverseCompany ( source : DisplayEmployee, view : InputSensor )
      var
            selection → Integer
            numberString, menuString → String
      make set of
            condition (source ≠ nil)
                  StringConversion(source.subNumber, numberString)
                  StringConcat(('name', numberString), menuString)
                  PopUpMenu ((selection, menuString), view)
                  condition (selection = 1)
                        StringSensor(source.name, view)
                  condition (selection ≠ 1)
                        TraverseCompany (source.subordinates[selection-1], view)
end
```