Evaluating The Mediator Method: Prism as a Case Study

Kevin J. Sullivan, *Member, IEEE*, Ira J. Kalet, *Member, IEEE Computer Society*, and David Notkin, *Member, IEEE*

Abstract—A software engineer's confidence in the profitability of a novel design technique depends to a significant degree on previous demonstrations of its profitability in practice. Trials of proposed techniques are thus of considerable value in providing factual bases for evaluation. In this paper we present our experience with a previously presented design approach as a basis for evaluating its promise and problems. Specifically, we report on our use of the mediator method to reconcile tight behavioral integration with ease of development and evolution of Prism, a system for planning radiation treatments for cancer patients. Prism is now in routine clinical use in several major research hospitals. Our work supports two claims. In comparison to more common design techniques, the mediator approach eases the development and evolution of integrated systems; and the method can be learned and used profitably by practicing software engineers.

Index Terms—Software engineering, design methodology, software evolution, integration, object-oriented, component-based, mediator, implicit invocation, abstract behavioral type, radiation treatment.

1 Introduction

A SOFTWARE engineer's confidence in the profitability of a new design technique depends to a significant degree on previous demonstrations of its profitability in actual, comparable practice. Trials of new techniques are thus of considerable value in providing factual bases for evaluating the risks and potential benefits involved in using new techniques in real projects with assets at risk. This paper provides such a basis for evaluating one recently proposed technique, the mediator method for developing and evolving integrated systems [42], [43]. An integrated system is one in which separate tools work together automatically to support the user. This paper presents a case study in which we used the method to develop and evolve an ambitious system, called Prism, for planning radiation treatments for cancer patients [19], [20], [21].

Our experience with the mediator method supports two claims. First, in comparison to common design techniques, our method eases the development and evolution of integrated systems. Second, it can be learned and used profitably by practicing software engineers. We observed that it takes experience and intellectual effort to become proficient with the method. It involves new ways of thinking about design that are not easily internalized.

- *K.J. Sullivan is with the Computer Science Department, Thornton Hall, University of Virginia, Charlottesvilles, VA 22903. E-mail: sullivan@cs.virginia.edu.*
- *I.J. Kalet is with the Department of Computer Science and Engineering, and the Department of Radiation Oncology, RC-08, University of Washington, Seattle, Washington 98195. E-mail: ira@radonc.washington.edu.*
- *D. Notkin is with the Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195. E-mail: notkin@cs.washington. edu.*

Manuscript received July 3, 1995; revised June 3, 1996.
Recommended for acceptance by J. Gannon.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95785.

Prism is now in routine clinical use in several major research hospitals, including those at the University of Washington, Emory University, and the University of Miami. In retrospect it would have been far more difficult to meet the requirements for Prism with the available time and resources had we not used the mediator approach. The rest of this paper discusses how the mediator concepts helped in the development and evolution of Prism. The paper is organized as follows. Section 2 introduces radiation treatment planning and Prism. Section 3 discusses both how common design techniques complicate the development and evolution of integrated systems, and how the mediator approach helps. Sections 4 through 9 present key parts of Prism. Section 10 discusses related work. Section 11 discusses the effort needed to build Prism and the code and documentation sizes. Section 12 concludes with a summary and with an evaluation of our experience with Prism as evidential support for our claims.

2 Prism

A radiation treatment planning (RTP) system is a collection of software tools for designing plans for treating cancer patients with radiation. A radiation treatment has to deliver enough radiation to damage a tumor without unacceptable damage to the surrounding tissues—such as the optic nerve or spinal cord. A satisfactory treatment plan defines a set of radiation beams and implanted radiation sources that delivers such a radiation dosage.

Designing radiation treatment plans is hard. It requires one to use radiographic images and other measurements to model patient anatomy and pathology; to model the shapes and directions of radiation beams and the capabilities of the machines that produce them; to calculate the radiation fields generated as high-energy particles interact with tissues; to visualize and iteratively refine treatment plans; and

to use hospital-quality data management methods. The complexity of the task makes computerized tool support indispensable.

Many RTP systems have been built [10], [25], [8], [35], including several at the University of Washington [18], [17]. However, many earlier systems have suffered from serious usability and software design problems. First, in many systems, the tools are not well integrated. In the first University of Washington (UW) system for example, planning functions were packaged as stand-alone tools that ran as separate processes loosely integrated through shared data files [18]. Second, many systems are tedious to use. In the second UW system, users have to traverse a broad, deep menu tree to access the planning functions [16], [17]. Third, many systems are hard to change. It was infeasible to integrate new AI-based tools into the second UW system, for instance, despite its object-oriented architecture [22], [30].

Prism was designed to overcome these problems. Ease of use demanded that tools be tightly integrated. In Prism, changing the patient anatomy or the position of a beam recalculates the radiation fields and updates all visualizations, for example. Usability also demanded that the system be flexible, dynamic, and broad in scope. Prism users can instantiate, place, and use any number of tools to create, edit, simulate, visualize, store, or retrieve treatment plans—usually with the click of a single button. Finally, the rapid advances being made in RTP, and the desire to use Prism both in production and as a research vehicle, demanded that the system accommodate continual evolution—especially the integration of new and the removal of old or unsuccessful tools—without structural decay.

Unfortunately, as Taylor and his colleagues had already observed, these requirements would pose a serious software development problem:

> "... [A] well-integrated environment is easiest to achieve if the environment is limited in scope and static in its contents and organization. Conversely, broad and dynamic environments are typically loosely coupled and poorly integrated. Unfortunately, poorly integrated environments impose excessive burdens upon users, and small static environments are quickly outgrown [44, p. 2]."

We developed the mediator method to resolve this conflict between *behavioral* integration on one hand, and the need for a well modularized software *architecture* [39] that would facilitate software development and evolution, on the other hand.

3 The Mediator Method

The benefits of the mediator approach are clearest when one compares the structures it produces with those produced by common techniques for designing integrated systems. We begin this section by briefly reviewing our earlier analysis of common approaches and how they unnecessarily complicate the structures of integrated systems. Then we use two simple examples from Prism to illustrate the mediator approach.

3.1 Shortcomings of Common Approaches

We have characterized four common techniques for designing integrated systems, which we call the *hardwiring*, *encapsulation*, *implicit invocation*, and *broadcast message server* (BMS) methods. All four approaches have in common that they represent key behaviors in the problem domain as corresponding components—tools or objects. The approaches differ in the way they integrate these components.

- In the *hardwiring* approach, one integrates components by having them call each other's operations. To integrate a model and a view object, for instance, one might augment the model to call the view. The problem with this approach is that it compromises component independence. Integrating a component into or removing it from a system generally requires changes to it and to the components that it interacts with; and the complexity of components increases with the degree of integration. In well integrated, dynamic systems, the structural complexity needed for behavioral integration is a major liability.
- In the *encapsulation* approach, one integrates components by wrapping them in an additional component that manages access to them and coordinates their behaviors. One could integrate a model and a view by defining a new model-view wrapper that aggregates and manages access to the underlying objects. One problem with this approach is that, while it preserves component independence, it compromises visibility. A component so integrated cannot be accessed except through the wrapper. When an existing component is integrated with with another, clients of the first component have to be changed to access it through the wrapper, complicating both the process of evolution and the structure of the resulting system.
- In what we call the *implicit invocation* approach, one integrates components by having them register directly with each other to receive event notifications. A view component might register with a model object, for example, and access the model then update itself when the model announces its events. The Smalltalk Model-View-Controller employs this strategy [24], [11]. The *Observer* pattern generalizes the idea [9]. Like hardwiring, this approach compromises component independence, since components have to register with and "understand" each other.
- The *BMS* approach is similar to the implicit invocation style, except that a distinguished component called a *broadcast message server* is interposed between invoking and invoked objects. A model object might send notifications to the BMS, and a view object might register with the BMS to receive model-generated events. The problem with the BMS approach is that even though components do not register with or call each other directly, they still have to "understand" each other. That is, the code for managing component integration is generally mixed in with the code for the components to be integrated, making it hard to independently add or remove components, to change how they work, or to change how they work together—how they are integrated.

3.2 The Mediator Approach

The basic idea in the mediator approach is to identify and modularize in separate components called *mediators* information about how other components work together. A mediator represents the *behavioral relationship* needed to integrate visible, independent components. By visible we mean that a component can be manipulated by other components at runtime. Visibility is important to integrating new components into an existing system without disrupting the clients of the existing components. By independent we mean that a component is developed, used, reused, and documented as a stand-alone unit. Independence is critical to simplifying all aspects of software development. The mediator approach comprises three design phases.

- First, you design the system architecture as a graph that we call a *behavioral entity-relationship (ER) model.* The nodes represent basic tool or component behaviors. The edges represent the behavioral relationships needed to integrate the basic behaviors. It is common for a single behavior to participate in multiple behavioral relationships. Hollingsworth and Weide would characterize this as the system or macro-architectural level of design [13].
- Second, you design an abstract component for each node and edge in the behavioral ER model. Our components are instances of what we call *abstract behavioral types* (ABTs). An ABT is like an abstract data type (ADT), but it exports events in addition to operations. Hollingsworth and Weide would characterize this as the component or micro-architectural level of design [13].
- Third, you implement the ABT components. Many object-oriented languages are suited to this task, but it may be necessary to implement an *implicit invocation mechanism* to support the concept of events in ABT interfaces.

3.2.1 Behavioral Entity-Relationship Modeling

We now consider each of these steps in more detail, starting with behavioral ER models. First, consider the part of Prism that models the set of tumors of a patient. The user of Prism can add and delete tumors, select them for editing, and view them in an overall patient visualization. Fig. 1 presents a behavioral ER model-based architecture of this part of Prism. The nodes (or components, in the lexicon of Garlan and Shaw [39]) represent respectively the behaviors of a dynamically changeable set of tumors, a multiple selection menu, a set of tumor editing panels, and a visualization panel.

The edges (connectors) denote relationships that integrate the component behaviors. The "bijection" between the tumor set and menu ensures that there is one menu button per tumor, even as the menu and tumor sets change. The second "bijection" ensures that there is an editor for each tumor designated by a *selected* menu button. The "injection" ensures that there is a graphic in the visualization for each tumor in the tumor set. The visualization may display graphics not represented in this subsystem—e.g., for the patient's normal anatomy.

Sections 4 through 6 elaborate in detail the behavioral ER modeling and design of this small but representative part of Prism. To set the stage for that discussion, we now narrow our scope to focus on an even simpler part of Prism: the *dialbox* user interface "widget."

The top of the panel in Fig. 2 presents three dialboxes in the context of the Prism tool for manipulating a single radiation beam. Each dialbox comprises a dial and a text line read-out. The user can change the angle of the beam relative to the patient by moving the dial with the mouse or by typing a new value. As a dial is moved, the text updates; and when the text is changed the dial moves. The dialbox can be seen as a paradigmatic, integrated system.

We define the architecture of the dialbox with a behavioral ER model—in this case one with two nodes and one edge. The nodes model the individual behaviors of the dial and text line. The edge models a relationship that maintains dynamically the equivalence of their values. This decomposition is critical, as it establishes the system modularization. In Prism we made our decomposition decisions based on informal but careful forethought about the structure and evolution of the application domain, the breakdown of the development task, and component reusability. In a larger-scale development, a more rigorous domain analysis might provide more effective guidance in behavioral ER modeling.

3.2.2 Designing with Abstract Behavioral Types

With the system architecture defined, we now realize each node and edge as an instance of a corresponding ABT. A key to the mediator approach is to realize the nodes of the behavioral ER model as *visible instances of independent ABTs.* Choosing an interface with operations and events that will support both the required component behavior and subsequent behavioral integration using mediators is the heart of the mediator approach.

Fig. 3 depicts our ABT interfaces. The left and right rectangles present the text line and dial as prototypical ABT instances, and the middle rectangle presents an instance of the mediator ABT. The figure as a whole presents a prototypical dialbox instance. The dial and text line ABTs export operations to set and get stored (string or numeric) values, and an event to signal changes in value. The events, like operations, have programmer-defined names and type signatures (parameters). Events are announced by operations whose effects satisfy the requirements for event announcement (e.g., changing the invoked object).

The operations exported by the mediator are responsible for updating the dial or text object when the other object changes. The mediator registers its operations with the dial and text line events (dotted lines). When the dial or text value is changed, the event notifies the mediator of the new value by invoking the registered operation. The operation computes a new value for the other object and calls the other object to update its value (solid lines).

This update causes a second event announcement, which recursively invokes the mediator. In Prism (a sequential system) we break the resulting circularity by having the mediator maintain a bit indicating whether an update is already in progress. When notified, the mediator checks the bit. If it is set, the mediator returns. If it is not set, the

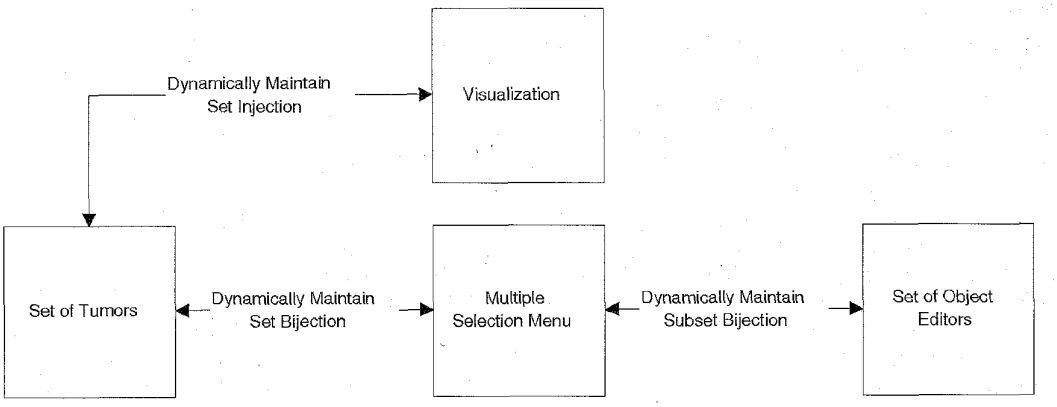


Fig. 1. Behavioral ER model for a representative fragment of Prism.

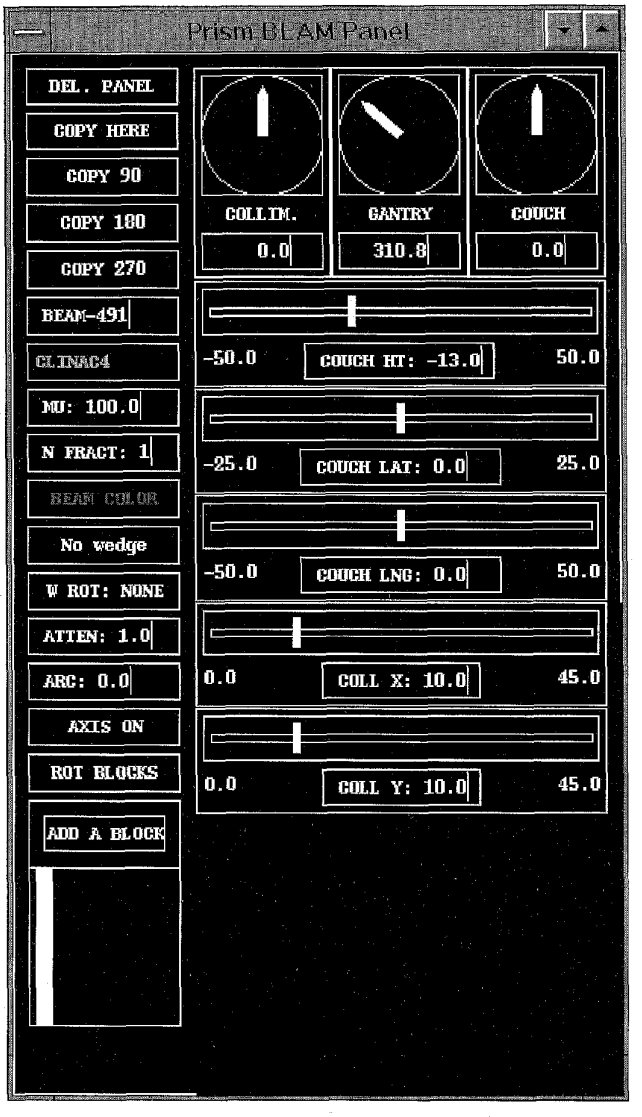


Fig. 2. A beam panel, with three dialbox widgets (in a row along the top) for controlling the orientation of the radiation beam and the patient couch.

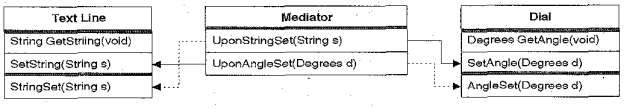


Fig. 3. The mediator design of the dial box system.

mediator sets it, performs the update, gets re-notified, dismisses the renotification, and then clears the bit just before completing its work. This example shows how we achieve integration while preserving the visibility and independence of the components.

3.2.3 Implementing Abstract Behavioral Types

Now, finally, we implement the ABTs. It is easy to implement ABTs in most object-oriented languages. One challenge is to simulate the events in object interfaces. With support for events in place, we then implement the independent ABTs and finally the mediator ABT. We address each of these tasks in turn.

Implicit Invocation. Event-based implicit invocation is a key part of the mediator approach. Fortunately, implicit invocation mechanisms can be added to many programming languages [28]. The *Observer* pattern provides one approach [9]. We designed our event mechanism to support the multiple events exported by ABTs. We implement each event as an attribute of the exporting ABT type—that is, as an event object instance variable. Riehle has recently cast this design concept in the form of an object-oriented design pattern [34].

Such an event object maintains a set of parties to be notified when the event is announced. It exports operations to add and delete these notifications and to announce the event. An operation responsible for announcing an event does so by calling the *announce* operation exported by the event object, passing to it the event parameters, such as the new value of the dial. External clients (mediators) register and unregister their operations by calling *register* and *unregister* operations exported by the event objects.

Fig. 4 presents our Common Lisp/CLOS [40], [1] implementation. We implemented events as association lists along with operations to manipulate them. The operations are implemented as macros. The notification set of an event

is the association list pairing the operations to be implicitly invoked with the objects to which they are to be applied. The event implementation supports three operations.

- *E.AddNotify(ob,op)* adds the pair consisting of the operation to be invoked *op* and the object (party) *ob* to receive the invocation. The receiving party is usually a mediator.
- *E.RemoveNotify(ob,op)* deletes the specified pair from the event's registration set.
- *E.Announce*(p_1, ... ,p_n) implicitly invokes all registered objects by iterating over the (*ob*, *op*) pairs and applying each operation *op* to the associated party *ob*. For each (*ob*, *op*) pair, *announce* calls *ob.op*(p_1, ..., p_n), where p_i are parameters compatible with the event type signature. In our Prism mechanism, a reference to the announcing object is passed as a parameter. The remaining actual parameters are supplied by the announcer. The operations invoked by the event must have signatures that conform to the event signature.

Independent ABTs. Implementing the dial and text line objects is now easy. Because the two are so similar, we just discuss the dial. We implement the dial as an instance of a CLOS dial class (see Fig. 5). The two key aspects of the design are the inclusion of an event as an instance variable, and the programming of the operation that changes the angle to announce the event. The announcement is done by a CLOS wrapper method around the *setf* operation that is used to change the value. When a client uses *setf*, the wrapper is called automatically. It erases the dial graphic, changes the angle value (within *call-next-method*), draws a new graphic, then announces the *angle-changed* event.

Mediator ABT. The mediator object in this case is the dialbox object itself. A dialbox thus aggregates instances of dial and text line classes and also makes them work together as they are manipulated directly. Fig. 6 presents our implementation, elided to suppress irrelevant details. The dialbox type initializer *make-dialbox* creates the text line and dial parts then registers its operations with their events. The code *ev:add-notify* registers the dialbox *db* to be notified by invocation of *upon-angle-changed* when the *angle-changed* event is announced. The event parameters identify the notified party (*dialbox*), the announcer (*dial*), and the new angle.

The mediator's *upon-angle-changed* operation first checks to see if an update is in progress. Finding none in progress, the mediator converts the numeric angle to a string and updates the text line. This text line announces its *new-info* event (not shown in the figure), which implicitly invokes the mediator's operation *upon-info-changed*. Now, finding an update in progress, the mediator just returns to the text line. The update of the text line completes and returns back to the mediator. The mediator then announces its own *angle-changed* event in order to notify any objects that view the dialbox system as a black box of the state change. Finally, the mediator clears its busy bit and returns. The original *setf* operation completes with the dialbox system now in a consistent state.

The dialbox announces its own event to notify any mediators that may consider it as a unit that the angle changed. This design reflects an interesting combination of traits: The dialbox is a mediator with respect to the dial and text line objects; and at the same time it acts as a more complex, independent object to clients who view it that way. It is an open system that ensures its own consistency in the face of operations on its parts; but it can also be used and treated as a black box within a larger aggregate. Indeed, while users operate on the dial or text line directly in Prism, the dialboxes are treated as black box abstractions by the rest of the beam panel.

4 Listing Objects in Selector Menus

The entire design of Prism is based on the mediator approach. In this and the following five sections we characterize the mediator design of Prism by showing how we applied the approach to representative parts of the system. We begin with what we call *selectors*, which we already introduced implicitly in Section 3.2.1.

A selector displays a menu whose buttons designate the elements of a set; and whenever a button is selected, the selector dispatches a tool *panel* for viewing and editing the object designated by the selected button. The first selectors that the user sees upon running Prism are those on the *patient panel*, presented in Fig. 7. This panel is the Prism "master control." It is used to create, load, and store patient cases and to select parts of patient cases for further manipulation. Here a patient case for one, "Joe Pancreas," has been loaded. The user selects objects for viewing and editing with the selectors at the bottom of the panel. The *organ selector* indicates that Joe's liver, spine, kidney, and external contour are modeled. The plan selector presents the names of three plans. One plan, "fixed grid," is selected for editing. A plan panel for that plan is thus displayed to the user (not shown).

Many Prism panels include selectors. Selectors are the Prism tool invocation mechanism. In the rest of this section we discuss how we structured the part of the selector responsible for keeping the menu consistent with the set of objects. In the next section we discuss how we dispatch tools by keeping a set of tool panels consistent with the subset of selected menu buttons.

4.1 Behavioral ER Model: Maintaining a Bijection between Sets

The basic requirement is that the overall set of buttons in a menu remain consistent with a set of objects in the face of changes to either set. If the user adds a button to the menu (by pushing the *Add* button on the selector), a corresponding organ (for example) has to be added to the set; and if an organ is added, a corresponding button has to be added to the menu.

The behavioral ER model for this system is a straightforward adaptation of that for the dialbox. We modeled the set of organs and the menu as separate behaviors. We modeled the menu as two sets of buttons: the set of displayed buttons, and the subset of selected buttons. The selector subsystem thus includes the set of objects, the menu, and a behavioral relationship that ensures that the set and menu remain consistent—that there is a bijective mapping between them. We were led to the behavioral ER model in Fig. 8.

```
(deftype event () 'list)
(defun make-event () nil)
(defmacro add-notify (party event operation)
  `(setf ,event
         (adjoin (list ,party ,operation)
                 (remove ,party ,event :test #'eq :key #'car))))
(defmacro remove-notify (party event)
  `(setf ,event (remove ,party ,event :test #'eq :key #'car)))
(defun announce (object event &rest args)
  (dolist (entry event) ; event is an a-list
          (apply (second entry) (first entry) object args)))
```

Fig. 4. Common Lisp/CLOS implementation of event object in Prism.

```
(defclass dial (frame)                          ; define a dial ABT
  ((angle :type single-float                    ; angle attribute
          :accessor angle
          :initarg :angle)
   (angle-changed :type ev:event                ; angle-changed event
                  :accessor angle-changed
                  :initform (ev:make-event))))

(defmethod (setf angle) :around (new-angle (d dial))
  (dial-erase-pointer d)                        ; erase old dial graphic
  (call-next-method)                            ; invoke inner wrappers
  (dial-draw-pointer d)                         ; draw new graphic
  (ev:announce d (value-changed d) new-angle)   ; announce value-changed
  new-angle)                                    ; setf must return value
```

Fig. 5. Key features of the implementation of the dial ABT.

```
(defclass dialbox (frame)                       ; the dialbox/mediator class
  ((the-dial :type dial :accessor the-dial)     ; references a dial
   (the-text :type textline :accessor the-text) ; and a text line,
   (angle-changed :type ev:event                ; and exports an event,
                  :accessor angle-changed
                  :initform (ev:make-event))
   (busy :accessor busy :initform nil)))        ; and avoids circularities

(defun make-dialbox (radius &rest other-initargs)
  (let* ((db (apply #'make-instance 'dialbox)))
    (setf (the-dial db)
          (apply #'make-dial radius :parent (window db))
          (the-text db)
          (apply #'make-textline width height :info "0.0" :parent (window db))
    (ev:add-notify db (angle-changed (the-dial db)) #'upon-angle-changed)
    (ev:add-notify db (new-info (the-text db)) #'upon-new-info)
    db)))

(defun upon-angle-changed (db ann val)
  (unless (busy db)                                 ; avoid circularity
          (setf (busy db) t)                        ;   "      "
          (setf (info (the-text db))                ; convert angle to
                (format nil "~5,1F" (mod val 360.0)))  ; string; update text
          (ev:announce db (angle-changed db) val)   ; announce dialbox event
          (setf (busy db) nil)))                    ; avoided circularity
```

Fig. 6. Key features of the implementation of the mediator for the dial and the text line.

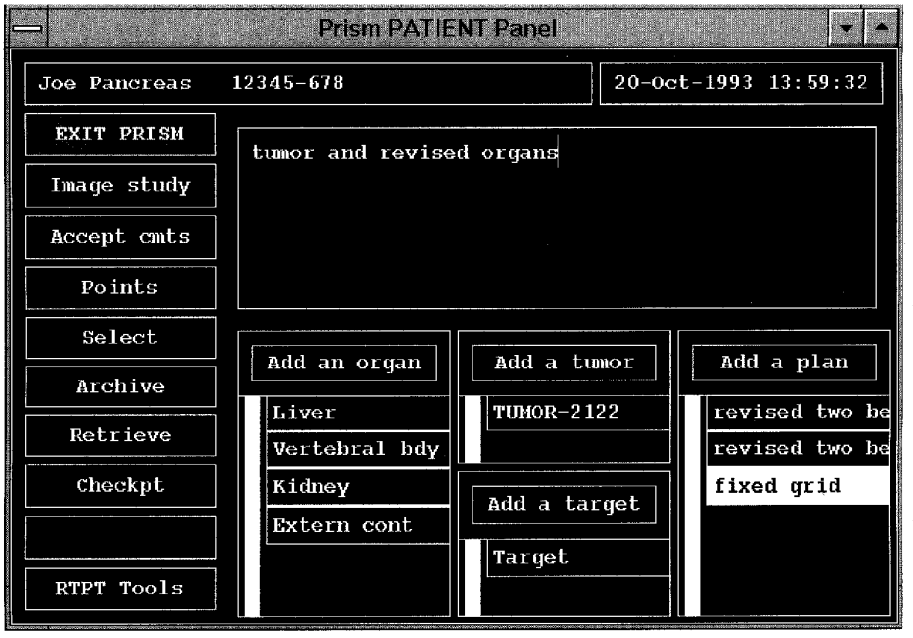


Fig. 7. The Prism patient panel.

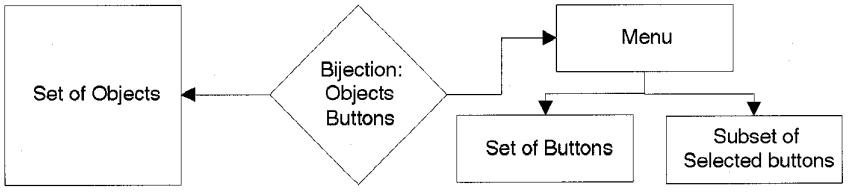


Fig. 8. Simplified behavioral ER model of the menu part of selectors.

The semantics of the sets conjoined with the invariant imposed by the relationship implies the propagation of component behaviors needed to effect integration. The visibility of the organ set and menu eases evolution by allowing new tools to operate on these components without requiring changes to the existing system. The independence of the types eases development by allowing the parts to be developed with minimal interaction. The separate and explicit representation of the semantically rich relationship also helps manage the complexity of the system by localizing the integration-related information.

Our examples are small but representative of the overall behavioral ER model of the Prism architecture. The whole Prism system is structured as a collection of visible, independent behaviors integrated in a network of behavioral relationships. This architectural style is largely responsible for the simplicity and adaptability of the Prism software, despite the high level of behavioral integration.

4.2 Design: Set ABTs and the Bijection Mediator

We implement sets as instances of a *Set* ABT with operations to insert, delete, and iterate over elements, and with events to indicate the successful insertion and deletion of elements. The operations are responsible for announcing events as necessary. The *insert(x)* operation announces *inserted(x)* if and only if x is actually added, which happens only if x was not already in the set, for example.

The menu ABT has operations to insert, delete, iterate over, select and deselect buttons, and events to signal insertion, deletion, selection and deselection of buttons. This design collapsed the interfaces of the two sets in the model of the menu into a single interface.

To simplify the presentation and to make a slightly more general point, we first treat the menu as if it were just a single set. See Fig. 9. (We discuss the grayed out parts of the figure in the next subsection.) The organ and button sets are *Set ABTs*—X and Y in the figure. The mediator references the sets and has operations to handle their insertion and deletion events. When the mediator is created, it registers with the appropriate events. When an element is inserted or deleted, the set announces a corresponding event (dashed lines), implicitly invoking the mediator. The invocations flows back along the dotted lines. The mediator calls the other set to update it (the solid lines), using a busy bit to avoid circularities.

A complexity not discussed above is the way the mediator handles deletions. When an element is deleted from a set, the problem is to find the element to delete from the other set. To do this, the mediator holds a relation between corresponding elements, which it maintains as the sets change. Now, when an element is deleted, the mediator looks up the corresponding element, deletes it from the other set, and then deletes the association from its relation.

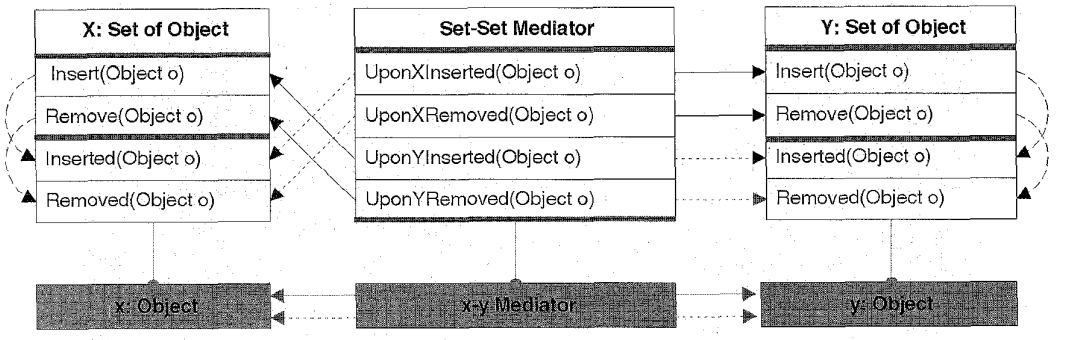


Fig. 9. The mediator design of the consistent sets system.

Representing multivalued attributes, such as the organs of a patient or the buttons of a menu, as *Set* ABTs is a cornerstone of the Prism design. It provides a runtime representation of dynamic entry and exit of elements into and from associations. Rather than having to maintain consistency in the face of language level instantiation and destruction of objects, we do so in the face of events indicating insertion into and deletion from sets.

4.3 Submediators: Integrating the Subcomponents

The grayed-out boxes in Fig. 9 reflect an additional requirement: When an organ and a button are associated by a selector, the name of the organ and the name on the button should be the same. If one updates the text of a button, the name of the corresponding organ should also change, and any change to the organ should be reflected by the button.

Behavioral ER modeling again helped us to develop a conceptual design, which we then implemented using ABTs. We define a new mediator type whose instances keep button and element names consistent. Then we design the "main" mediator (between the sets) to deploy and retract these "submediators" as necessary. When an organ is added to the organ set, the main mediator adds a corresponding button to the button set, updates its own relation, and creates a submediator to keep the organ and button names consistent. The objects representing the tuples in the relation could themselves serve as the submediators.

We use this approach to keep components of structured aggregates consistent with their counterparts in other such aggregates, without skewing the designs of either the components or the aggregates. The ability to integrate objects for the duration of their association in a relation shows the leverage provide by separate, explicit representation of behavioral relationships as mediators.

5 Using Selectors to Dispatch Tool Panels

In addition to displaying the contents of sets, selectors also allow objects to be selected for viewing and editing in new panels. One selects the plan fixed grid, for example, by pressing the button so labeled on the plan selector of the patient panel. This action highlights the button and launches a plan panel. The plan panel itself includes selectors that display and allow selection of the parts of a plan. A plan includes a set of radiation beams, a set of views of the treatment plan, and other object sets. Selecting one of the views from the view selector would launch a new view panel.

Fig. 10 depicts one view panel. This particular panel presents a transverse view. The background displays a radiographic image. The contour lines in the foreground outline the organs in the patient model (kidneys, liver, spine, and skin), the tumor and enclosing region to be irradiated, and the two radiation beams defined in the selected plan (one entering from the lower right, the other from the upper left).

5.1 Behavioral ER Model: Another Bijection

The behavior that we require of the selectors, then, is a dynamically maintained association between the subset of selected menu items and a set of editing panels. When a button is selected (highlighting it), a panel is dispatched. When the panel is closed the button is deselected. Fig. 11 presents the behavioral ER model for the overall selector subsystem. This model extends the one we presented above. We have added a set of panels and a new relationship. The new structures are colored gray in the figure.

The new relationship requires the panel set to be consistent in a bijective sense with the subset of selected buttons. The connection from the new relationship to the old one indicates that the new one "uses" the old. Specifically, it uses the relation between buttons and organs that is maintained by the old relationship to determine which object corresponds to a selected button. That information is needed to associate a new panel with the object designated by the selected button.

With the selector problem solved once, we applied the solution to all selectors in Prism. The same code, parameterized to create different kinds of panels for different kinds of set elements, is shared by all selectors.

5.2 Design: Reusing Sets and the Bijection Mediator

To accommodate the change in requirements reflected in the extension of the behavioral ER model, we add another *Set* ABT and another bijection mediator to the existing selector subsystem. We do not change existing modules. Fig. 12 details the extended design.

Specifically, we add a set of panels and a mediator between it and the menu. (The figure presents a more complete menu ABT interface in this example.) When a button is selected, the new mediator reads the relation maintained by the old mediator, as depicted by the curving arrow, to find out which object is designated by the selected button. The mediator then creates a new panel, attaches it to the designated object, and adds the panel to the panel set.

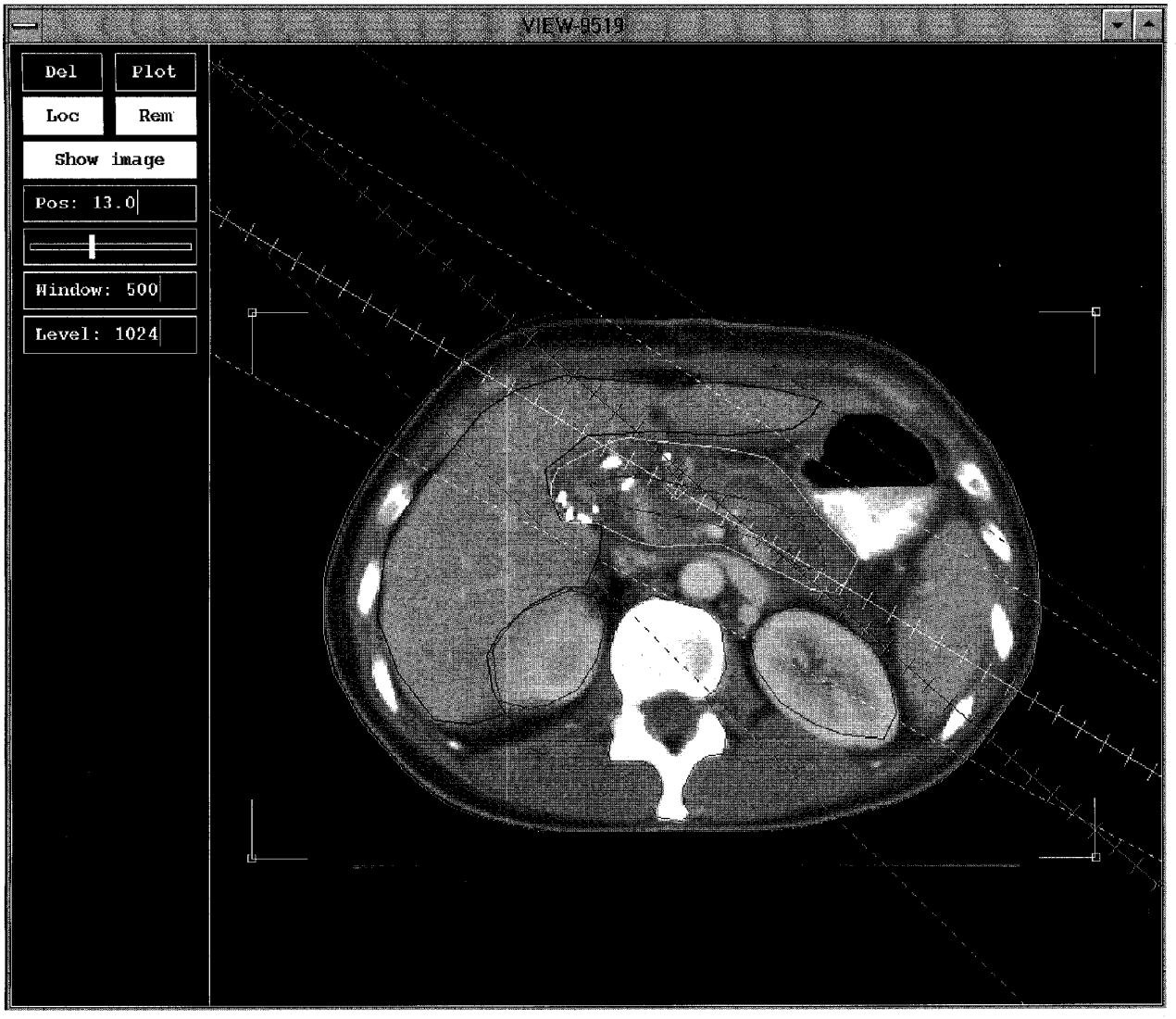


Fig. 10. A transverse view of a Prism radiation treatment plan.

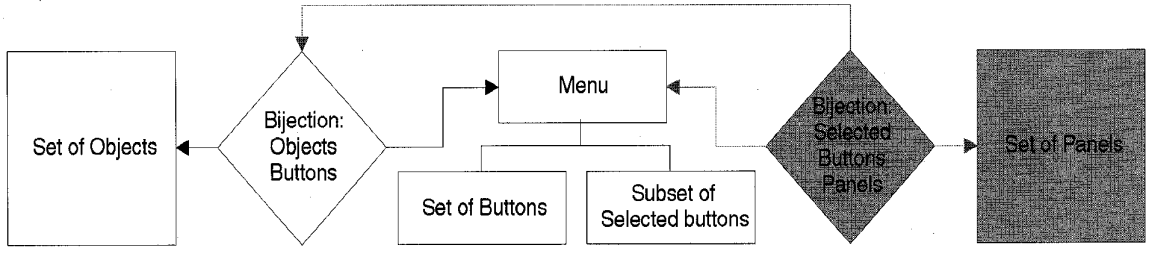


Fig. 11. Simplified behavioral ER model of the selector subsystem.

The mediator approach transports the ease with which we extended the behavioral ER model to the design and implementation levels. Extending the implementation in parallel with the behavioral ER model—integrating new parts without disrupting the existing design or implementation—is the way we constructed and continue to evolve Prism.

The preceding examples illustrate how we apply the approach at multiple scales, or granularities. We viewed the dialbox as an independent system to be integrated into the beam panel. We viewed the set of panels as an independent system to be integrated with the menu. We then shifted our perspective to the selector as an independent system to be

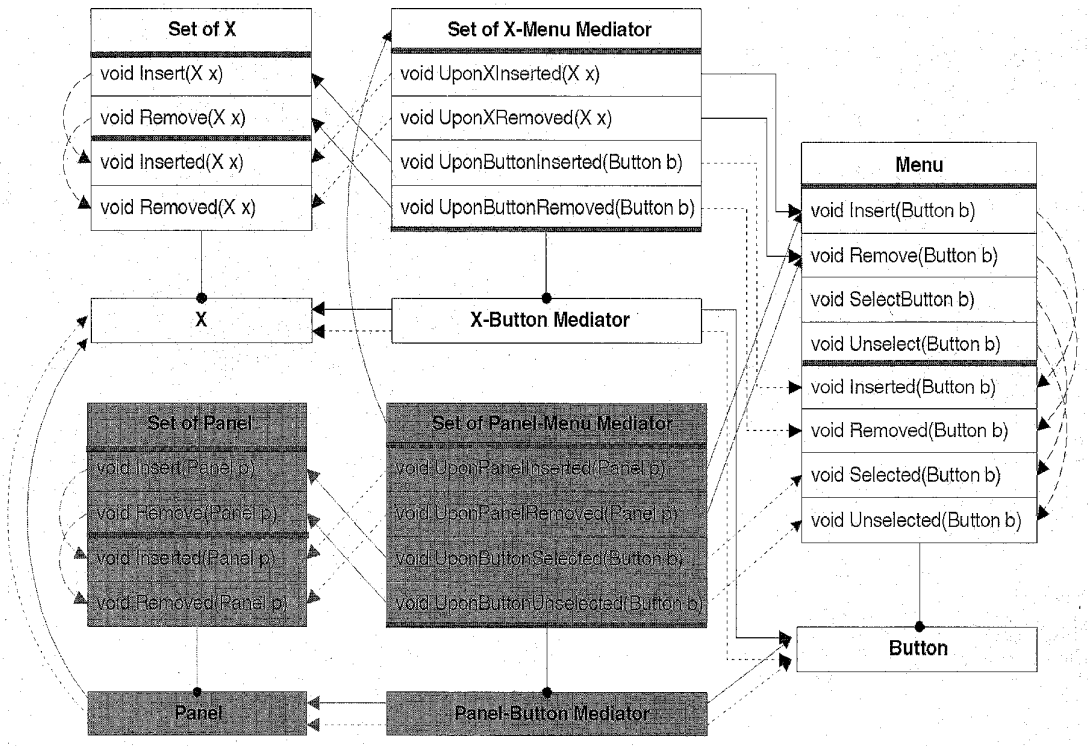


Fig. 12. Mediator design of the selector subsystem.

integrated into the plan and patient panels (as well as some others). The patient panel is in turn integrated with the overall patient case object (the details of which we do not discuss in this paper). The decomposition of behaviors at all granularities into independent behaviors integrated by mediators provided us with real intellectual and managerial leverage. This sort of decomposition serves the essential purpose of modularization, after all: to conquer by dividing *effectively* [7].

6 Keeping Graphical Views Consistent

We now complete our elaboration of the behavioral ER model presented in Section 3.2.1 by discussing the integration of object sets with views. Fig. 13 illustrates the four kinds of views that Prism supports: orthographic projections along the axes of the "patient coordinate system," and (in the lower left) perspective views taken from the origins of radiation beams. The user can define any number of views, and can have any number of view panels active.

The aspect of integration we address in this section is the need to keep views consistent as the patient case changes. Specifically, we wanted to display each organ, tumor, target, point, dose distribution, etc., in each view. We want to keep views consistent as objects are added to, deleted from or changed within the patient case, and as views are added and deleted. Adding an organ, for example, should be reflected in each view; adding a view should cause each organ to be rendered in the view; and changing an organ should cause the corresponding graphic in each view to be updated as necessary.

6.1 Behavioral ER Model: Maintaining a Cross-Product

Given a set of objects and a set of views, we saw that we needed a mediator between the two sets. This mediator would use submediators to keep each object consistent with a graphic depicting it in each view in the set of views. Thus, one such submediator would be needed for each element in the *cross product* of the object and view sets. We modeled the architecture of the view system with the behavioral ER model in Fig. 14. One of these mediating structures is needed between each object set in the overall patient model (organs, tumors, etc.) and the set of views. Note that we are able to ignore the participation of the object set in other behavioral relationships, such as the one between the object set and the menu part of the selector for the set. In this way, the mediator approach achieves an effective separation of concerns.

6.2 Design: The Cross-Product Mediator

The mediator in this model is similar to that in the selector subsystem, but instead of computing and maintaining a bijection, it maintains the cross product relation. As elements are added to and deleted from either set, the mediator updates the cross product, inserts or deletes graphics as necessary, and dispatches submediators to keep individual objects consistent with their graphical depictions.

These submediators are complex. They embody design decisions about how to render objects. This design separated the implementation of the graphics pipeline from the code responsible for deploying the submediators. This modularization paid off when Kalet discovered that his

initial design for rendering views was unworkable. The problem was in rendering multilayered images in the X windows system [37]. The specific difficulty was in incrementally updating renderings of contours drawn over background radiographic images.

The mediator architecture that isolated the graphics code in submediators enabled Kalet to fix the problem with no impact on the superstructure that actually deployed the submediators. We did not have to change or even consider in detail the organ set, view set, or the mediator between the organ set and set of views. Also insulated from the change were the organs themselves, the mediator between the organ set and the menu in the organ selector, between the menu and the panel set, and so on. The mediator architecture effectively separated these concerns, allowing Kalet to correct the problem with confidence that the changes would not disrupt the rest of the system.

7 Integrating Multiple Views

In this and the next two sections, we discuss additional aspects of the Prism design. In this section we address the user's understanding of spatial relationships among different graphical views.

With many orthographic views, a problem for the user is to understand how views relate to each other. Along what line does a transverse view intersect a coronal view? In this section, we discuss how we enriched the viewing system by depicting the intersections of the viewing planes of orthographic views in each orthographic view. We did this to help the user to mentally integrate two-dimensional views into a three-dimensional mental model of a plan. Specifically, each orthographic view presents a set of graphics called *locators*, each indicating how the given view intersects with the viewing plane of another orthographic view.

A locator is a displayed as a white line, illustrated by the "crosshairs" in Fig. 13. The vertical locator in the transverse view represents the intersection of that view with the sagittal view. The horizontal locator in the transverse view represents the intersection with the coronal view. The horizontal locator in the coronal view indicates the intersection with the transverse view; and the vertical locator represents the intersection with the sagittal view. Mentally rotating a view around one of its locators relates the given view to the view corresponding to the locator.

Views and locators are tightly integrated. Locators are created and deleted as views are added to and deleted from the view set, and each locator must remain consistent with the viewing plane of the view it depicts. So, creating a new view adds locators to existing views; moving the viewing plane of a view updates the corresponding locators in other views; and moving a locator (using the mouse) updates the viewing plane of the other view, and hence also updates locators that depict that view in all other intersecting views.

7.1 Behavioral ER Model: The Intersects Relation on Views

In this situation, behavioral ER modeling led us to decompose the system into a set of views, a set of locators, and a relationship to keep the sets and their elements consistent. The question, again, was what relation does the "main" mediator maintain? Rather than a bijection or cross-product, the key relation here is the symmetric intersection relation over orthographic views. We want one locator for each intersection. We want that locator to be displayed in the intersected view. And we want it to remain consistent in position and orientation with the intersecting view. Fig. 15 presents a behavioral ER model representing this conceptual design.

7.2 Design: The View Intersection Mediator

The design for this subsystem is analogous to the designs discussed above. A mediator maintains a relation between aggregates, deploying submediators to integrate associated elements. These submediators are identical, but for details, to mediators already discussed. We optimized the design by dispensing with the set of locators. The main mediator just adds locators directly to the sets of objects displayed in the views.

Computing the intersection relation is simple. Only orthographic views are relevant, and two such views will intersect if and only if their types (i.e., transverse, sagittal, coronal) differ. These types are encoded in attributes of view objects, so only attribute comparison is needed to determine an intersection of a new view with views already in the view set.

Again, the mediator approach enabled the development and integration of this machinery independently of that for other roles in which views participate: involvement with the views selector; the relationships linking views to the elements they display; and so on.

8 The Beam Panel and its Collimator Subpanel

We now briefly consider a different sort of mediator—one that keeps one part of a user interface consistent with a selection in a different part of the interface. The beam panel (see Fig. 2) has this behavior. The dialboxes at the top of the panel rotate the couch, the gantry carrying the treatment apparatus, and the collimator that shapes the radiation beam. The sliders in the middle of the panel adjust the lateral and longitudinal positions and height of the couch. The remaining sliders comprise the *collimator subpanel*, which is used to set the collimator opening.

Different makes and models of treatment machines have different kinds of collimators. Changing the kind of machine designated as generating a radiation beam may imply a change in collimation system, requiring a change in the collimator subpanel. A Clinac-4, for example, requires two sliders to adjust its two "jaws." Changing to a machine with a collimator that has more jaws requires more sliders. Changing to a "multileaf collimator" (a kind of collimator we discuss further in Section 9) requires a different collimator interface.

8.1 Why We Chose Not To Inherit

One question when designing this part of the system was whether to use inheritance to model different kinds of beams and beam panels. We first tried defining a beam class with

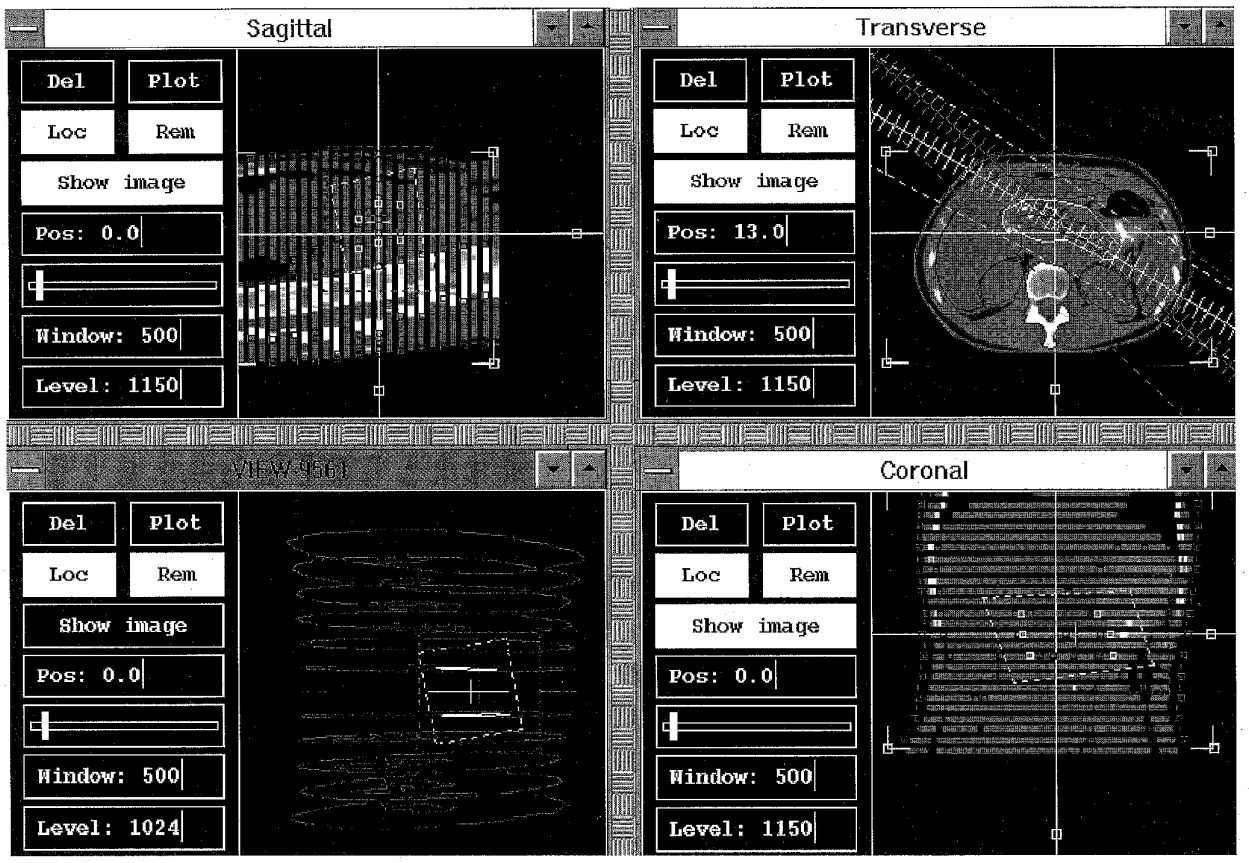


Fig. 13. Four view panels: transverse, coronal, sagittal, and beam's eye.

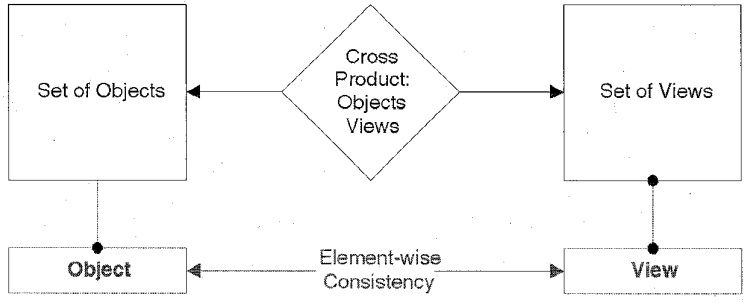


Fig. 14. Behavioral ER model for the core of the Prism viewing system.

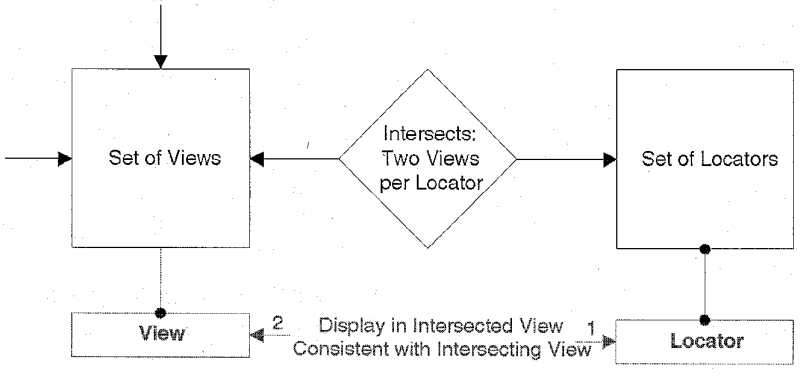


Fig. 15. Behavioral ER model of the Prism locators subsystem.

subclasses specialized first by the kind of collimator used, and then by other parameters (such as particle type). The result seemed arbitrary. Why specialize first by collimator type then by particle type? We tried other orders, too.

This approach had two problems. First, every specialization ordering seemed artificial because the specialization dimensions are independent. The beam "types" occupy a grid not a hierarchy. Second, changing a beam's machine attribute would have required dynamically changing the class of the beam being edited and dynamic replacement of the beam panel doing the editing. Although CLOS does support dynamic type conversion, we found that mechanism to be too complex for such a straightforward concept.

8.2 A Mediator Dynamically Selects the Right Part

Instead, we defined a single beam class with attribute values indicating specializations in particle type, collimation system, and other dimensions. We then used inheritance to model different kinds of collimators. To change the collimator type of a beam, we assign a collimator object of a different subtype to the collimator attribute. We modeled the beam panel in the same way, with a collimator subpanel as an attribute.

This made it easy to maintain the correspondence between machine type and collimator subtype within the beam, and to integrate the beam and beam panel. When the machine type changes, the collimator object in the beam is easily replaced. When the type of collimator changes, the beam announces an event. A mediator linking the beam and the beam panel responds by replacing the collimator subpanel.

9 Accommodating Requirements Evolution

Thus far, we have discussed aspects of Prism that were specified by Kalet before he knew of the mediator design approach. It is fair to take such requirements—clearly not contrived to yield predetermined outcomes—as a reasonable test of whether the mediator method eases the development of integrated systems like Prism. Kalet's success in meeting the Prism requirements in a timely manner and on a modest budget provides strong evidence for our claim that the approach overcomes the serious structural problems that plague common approaches. Our approach eases the development of integrated systems like Prism—rich in function, tightly integrated, sequential in execution.

What we cannot as conclude quite as confidently based on the initial effort is that the mediator approach eases software *evolution* in the face of changes in requirements not foreseen by the specifier but that arise in the course of the actual use of the system [26]. Of course there is no way for any approach to reliably accommodate entirely unforeseen requirements changes. However, we constructed the mediator method to accommodate key kinds of changes that are characteristic of integrated systems—namely the addition, deletion and modification of new behavioral entities and relationships.

The question is, does it work for behaviors and relationships that were not anticipated? The data are limited, and it is easy to construct situations in which change is hard. Nevertheless, our actual experience with Prism has been positive. We conclude our technical discussion of Prism with a brief description of one such unanticipated change and the ease with which the mediator architecture accommodated it.

The change was the integration into Prism of a new tool to handle leaf collimators. A leaf collimator has many independently adjustable leaves that provide flexibility in shaping a radiation beam. Two machines with leaf collimators are in use at the University of Washington Cancer Center. One is a cyclotron generating neutron beams. The other is a more conventional treatment machine generating X-rays and electrons.

The collimator subpanel of the new beam panel has a button called leaf display that brings up the leaf panel. The leaf panel shows a beam's eye view, the desired beam's portal contour (i.e., the shape of the hole through which the beam is to pass), and one text line for each leaf setting. The contour is drawn with the beam portal editor, which is not discussed here, but which reuses the same contour editing function used to model organs. A white, zig-zag line appears showing the best fit of the collimator leaves to the specified portal for the current rotational angle of the collimation device. If the collimator angle is changed (e.g., by a dialbox on the beam panel), the white "leaf-setting" shapes change to show the best fit for the new angle, and all the leaf text lines update, as well.

The leaf panel was added in the period between clinical deployment, July of 1994, and November of 1994. It required no modifications of the beam object, the beam panel object, the beam graphics, the beam's eye view graphics, the leaf collimator object, or anything else, except to add a button to the collimator subpanel and have the button call the leaf panel constructor. The panel code itself is small since it reuses a lot of the machinery of the system, including the beam's eye view. It is integrated with the collimator angle by registering a small function (eleven lines of Common Lisp code) with the existing *new-coll-angle* event of the beam object.

10 Related Work

The benefits of the mediator approach appear to be significant, yet none of the basic elements of the approach is novel. Rather, the novelty and utility are largely in the careful engineering of macro- and micro-architectural elements into a combination designed to ease the development and evolution of a broad class of integrated systems. In this section we briefly compare and contrast our work with some key related work.

The notions of separate representations of relations, and of the benefits of making such separations, are well established. Chen's entity-relation data modeling approach [5] established this pattern, albeit for the purpose of data rather than behavior modeling. The Object Modeling Technique of Rumbaugh et al. was perhaps the first major work to introduce it into object-oriented design [36]. In contrast to our work, however, the ability of Rumbaugh's relations to respond to the activities of the components they relate is limited to propagation of operations. Designer-specified ABT events allow us to implement more complex integration semantics.

Specification of behavioral relationships has also received attention. The ISO General Relationship Model [15] provided syntax for specifying relationships with behaviors. Helm et al. devised *contracts* to specify the obligations of objects that interact with each other [12]. In contrast to the ISO work, we do not present a specification notation. Our notion of behavioral relationship differs from contracts in that contracts state how given components call and notify each other, whereas our behavioral relationships require objects to work together without structural dependencies.

Implicit invocation has also received much attention as a software design and programming construct. An earlier paper describing the basic mediator concept [42] provides a survey, taxonomy and a semi-formal model of the design space for implicit invocation mechanisms, as they have appeared in a variety of domains, including database and artificial intelligence programming. Implicit invocation has been used as well for tool integration [33], consistency management of user interfaces [24], and general procedural programming [28]. Formal models of implicit invocation mechanisms [38], the codification of implicit invocation in numerous design patterns, and the support for the construct provided by technical architectures such as OLE [3] and CORBA [29] attest to the widespread recognition of implicit invocation as a key construct for structuring software systems.

Nor are class interfaces that export events new. ISO module specifications [14] include events. TICKLE events are explicit in interfaces [6]. There are many other examples. One of our concerns was the usability of our method by engineers trained in abstract data type and classical object-oriented design, which led us to present the ABT as a straightforward generalization of the familiar ADT, with events defined as "dual" to operations.

The mediator concept also has precedents. Objects that make other objects interact is the heart of Borning's ThingLab constraint programming system [2]. The structure and semantics of ThingLab objects are quite restricted, however. Wiederhold has developed a rich mediator concept [45]. Wiederhold's mediators essentially provide intermediate abstractions that insulate workstation-based applications from the complexities of heterogeneous, distributed data sources. Perhaps the key distinction from our work is that applications "know about" and use Weiderhold's mediators directly, whereas our mediators integrate stand-alone objects that do not "know about" the mediators that integrate them with other objects.

The Gamma et al. *Mediator* design pattern [9] is similar to our mediator. However, the Gamma et al. *Mediator* pattern requires the mediated objects to call the mediators. The ease of software development and evolution afforded by our approach depends on the use of mediators to integrate structurally independent objects. Our mediator concept both anticipated and can be emulated by a combination of the Gamma et al. *Mediator* and *Observer* patterns. However, there is more to our approach than that combination. The observer pattern does not suggest the ABT module construct as a micro-architectural building block. Nor does it address the macro-architectural issues covered by behavioral ER modeling, for example.

11 Development Effort and System Size

Our application of the mediator approach to Prism has clearly been profitable. To give a sense of the scale of Prism and of the effort that produced it, we present detailed figures.

Prism has grown from about 18,000 lines of Common LISP and CLOS, 4,500 lines of Pascal, and 11,000 lines of LaTeX documentation in September, 1993 (when the core system was completed), to about 43,000 lines of Common Lisp and CLOS, the same 4,500 lines of Pascal code, and 23,000 lines of LaTeX documentation.

The Lisp code handles modeling, visualization, and file management. The Pascal, adapted from an earlier system, computes dose distributions. Of the Lisp, about 8,300 lines now handle user interface widgets, sets, relations, and events. Prism model objects take 6,200 lines; mediators take 2,200 (excluding code in panels, which include the "object-panel" mediators); panels take 9,500; graphics take 3,500; and the rest includes file input and output and other miscellaneous functions.

The code density is about 30 characters per line, with blank lines and concise documentation text included. In comparison to our core system, Kalet's first system has 47,000 Pascal lines. Kalet's second system has 41,000 lines of Pascal, 5,000 for dose distributions. Comparing with another system, the basic functions taking 18,000 lines in the original core system required about 60,000 of C [23] and C++ [41] in GRATIS [35], of which about 14,000 are for interface widgets. Those functions taking 4,500 lines of Pascal in Prism, take about 12,000 lines of C in GRATIS.[1] Prism is small and clean relative to its function.

Prism was built on a modest budget in person-hours and with a small project team. The effort lasted from January, 1990 to present. Eleven people were involved at different times. The total effort to build the core was about five person years. Out of this total, requirements specification (done before the collaboration between Kalet and the developers of the mediator approach began) took 24 person months. Design and implementation, which were the focus of the collaboration, took 20. Developing electron beam dose calculation code took 11. The total effort expended to date is just under eight person years.

Prism is portable. It runs without source code modification using Allegro Common Lisp (CL) and Lucid CL on Sun Sparcstations (2 and 10). It runs using Allegro CL on DECstation 5000, IBM RS6000, Silicon Graphics Indigo, HP9000 series 700 workstations, and on DEC Alpha machines. The Prism Pascal code is ISO level 0 compliant and runs without modification on all of the above systems.

Prism performs adequately on high-end workstations. We use HP9000 series 700 workstations for development and production. The costliest operations by far are rendering pictures with projected three-dimensional graphics and computing background images. Background image display can be turned off in a given panel for faster response. Updates to interface widgets—menus, buttons, etc.—are comfortably fast. Event registration, unregistration, and announcement

1. Personal communication with Gregg Tracton, Department of Radiation Oncology, University of North Carolina.

are performed many times, but the cost in memory and CPU is negligible in comparison with other functions.

12 Conclusion: Prism as Evidence

The collaboration that began when Kalet (Radiation Oncology) decided to use the mediator method of Sullivan and Notkin (Computer Science and Engineering) has succeeded. Kalet and his colleagues now routinely apply behavioral ER modeling and design. Prism implements Kalet's original specification. It is in clinical use as the primary radiation treatment planning system at several cancer centers. It continues to evolve. It is rich in functioning compared to related systems; and despite tight integration, it has retained architectural integrity and flexibility. The system is a good platform not only for treatment planning at present and into the future, but also for software engineering research on issues of integration, architecture and software evolution.

Many factors were critical to the success of the project. One was Kalet's aversion to "gold plating" during requirements definition. It is possible that using Common Lisp was critical. The expertise of Kalet and his team of developers, built up over two previous RTP system building efforts, was indispensable. So, how can we assess the impact of the mediator approach?

The mediator approach permitted us to achieve behavioral integration while averting the structural problems inherent in common approaches to designing integrated systems—problems that foster all manner of derivative difficulties, in design, debugging, testing, understanding, evolution, and so on. Prism thus serves in some sense as a case study in the proper anticipation and avoidance of a certain important kind of failure [32].

Nevertheless, we do not present our conclusions as *truthful and rigorous findings*, which Brooks defines as, "results properly established by soundly-designed experiments and stated in terms of the domain for which generalization is valid [4]." Rather, we submit them as *useful rules of thumb*, "generalizations, even those unsupported by testing over the whole domain of generalization, believed by the investigators willing to attach their names to them [4, p. 2]."

Why, then, do we believe our rules of thumb? First, we can evaluate the artifact and the approach that produced it using widely accepted design criteria, such as modularity [7], information hiding [31], modular continuity [27]. Prism is clearly factored into independent parts integrated by separate mediators; and it was the mediator approach that produced this modularization. We have presented but a few parts of Prism, but parts that are representative of the whole Prism architecture.

Second, we can ask the client Kalet (admittedly a coauthor of this paper) whether the approach helped significantly. Kalet's basic claim is that without the conceptual and structural benefits of the mediator approach, meeting the functional requirements for Prism would have taken far more resources than were spent or even available; and the resulting system architecture would have been overly complex, hard to develop, and inflexible with respect to the ongoing integration of new capabilities.

This assessment is not the uninformed opinion of a novice designer in an unfamiliar application domain. Kalet and colleagues built two previous RTP systems [18], [17]. These systems have seen extensive clinical use. The second system was designed using object-oriented techniques, as described in the archival computer science literature [16], [17]. And the difficulties with the evolution of the system built using the common techniques are well documented [30].

Finally, as software engineering researchers, we can evaluate whether the mediator method helped make the clients more effective software designers. Our evaluation is that it did, but the transition was harder than expected. The mediator approach represents a significant change in how one models, designs, and implements systems. Just as the transition from a structured to object-oriented style requires time and a visceral understanding, so does shifting from the common object-oriented techniques to behavioral ER modeling and mediators. Superficially, one can believe this is a small shift. Our experience with several members of this project and with students and other colleagues indicates that the shift is significant (even though the method can be characterized as a variation of object orientation). Proficiency with the approach requires judgement and insight that is gained through repeated, practical application of the basic concepts over a substantial period of time.

Brooks identifies as a major issue facing the research community the tension between "narrow conclusions proved convincingly by statistically sound experiments and broad conclusions generally applicable, but supported only by unrepresentative observations." Brooks states that, *"Over-generalized findings from other designers' experiences are more apt to be right than the designer's uninformed intuition* [4, p. 2]." The careful, aggressive application of new techniques to demanding projects and the engineering evaluation of the resulting experiences provide both the researcher and practitioner with an indispensable basis for evaluating new design techniques. Our evaluation of the mediator approach, while not scientifically rigorous, does provide a substantial factual basis for assessing its problems and promise.

Acknowledgments

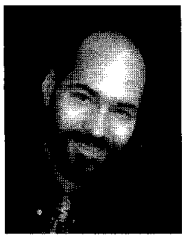
Jonathan Unger kindly provided a great deal of technical assistance. Thanks to Matt Conway for pointing out in his dissertation proposal the referenced SIGCHI article by Fred Brooks. This work was supported, in part, by Grant No. R01 LM04174 from the National Library of Medicine; Contract No. N01 CM97566 from the National Cancer Institute; by a grant from General Electric Medical Systems; by the National Science Foundation under Grants CCR-9113367, CCR-8858804, CCR-9502029, and CCR-9506779; and by SRA (Tokyo).

References

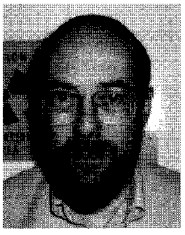
[1] D.G. Bobrow et al., "Common Lisp Object System Specification X3J13, Document 88-002R," *ACM SIGPLAN Notices* 23, Sept. 1988.

[2] A. Borning, "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory," *ACM Trans. Programming Languages and Systems* 3, vol. 4, pp. 353–87, Oct. 1981.

[3] K. Brockschmidt, *Inside OLE*. Redmond, Washington: Microsoft Press, 1995.
[4] F.P. Brooks, "Grasping Reality Through Illusion—Interactive Graphics Serving Science," Plenary Address, SIGCHI Bulletin, *Conf. Proc., Human Factors in Computing Systems*, pp. 1–11, May, 1988.
[5] P.P. Chen, "The Entity-Relational Model—Toward a Unified View of Data," *ACM Trans. Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976.
[6] T. Collins, K. Ewert, C. Gerety, J. Gustafson, and I. Thomas, "TICKLE: Object-Oriented Description and Composition Services for Software Engineering Environments," *Proc. Third European Software Eng. Conf.*, Milan, Italy, pp. 408–423, Oct. 1991.
[7] E. Dijkstra, "Programming Considered as a Human Activity," *Proc. 1965 IFIP Congress*, Amsterdam, The Netherlands: North-Holland, 1965, pp. 213–217, E.N. Yourdon, ed., *Classics in Software Eng.*, pp. 3–9, New York: Yourdon Press, 1979.
[8] B.A. Fraass and D.L. McShan, "3-D Treatment Planning I, Overview of a Clinical Planning System," I.A.D. Bruinvis, P.H. van der Giessen, H.J. van Kleffens, and F.W. Wittkamper, eds., *Proc. Ninth Int'l Conf. Use of Computers in Radiation Therapy*, pp. 273–277, Amsterdam: North-Holland, 1987.
[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.
[10] M. Goitein et al., "Multidimensional Treatment Planning: II, Beam's Eye-view, Back Projection, and Projection Through CT Sections," *Int'l J. Radiation Oncology, Biology and Physics*, vol. 9, pp. 789–797, 1983.
[11] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading, Mass: Addison-Wesley, 1983.
[12] R. Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *Proc. OOPSLA/ECOOP 90*, pp. 169–180, 1990.
[13] J.E. Hollingsworth and B.W. Weide, "Micro-Architecture vs. Macro-Architecture," Ohio State University Technical Report OSU-CISRC-11/94-TR57, Nov. 1994.
[14] ISO/IEC JTC/SC21, "ISO/IEC IS 10165–1: Information Technology—Open Systems Interconnection—Structure of Management Information—Part 1: Management Information Model," June, 1991.
[15] ISO/IEC JTC1/SC21/WG4, "General Relationship Model—Third Working Draft: ISO/IEC JTC1/SC21 N 7126," May, 1992.
[16] J.P. Jacky and I.J. Kalet, "An Object-Oriented Approach to a Large Scientific Application," *Proc. Conf. OOPSLA '86 Object Oriented Programming Systems, Languages and Applications.*, N. Meyrowitz, ed., pp. 368–376, 1986.
[17] J.P. Jacky and I.J. Kalet, "An Object-Oriented Programming Discipline for Standard Pascal," *Comm. ACM*, vol. 30, no. 9, pp. 772–776, Sept., 1987.
[18] I. Kalet and J. Jacky, "A Research-Oriented Treatment Planning Program System," *Computer Programs in Biomedicine*, vol. 14, pp. 85–98, 1982.
[19] I. Kalet, J. Jacky, S. Kromhout-Shiro, B. Lockyear, M. Niehaus, C. Sweeney, and J. Unger, "The Prism Radiation Treatment Planning System," Technical Report 91-10-03, Radiation Oncology Dept., Univ. of Washington, Seattle, Washington, Oct. 31, 1991.
[20] I. Kalet, J. Unger, C. Sweeney, S. Kromhout-Shiro, J. Jacky, and M. Niehaus, "Prism Graphical User Interface Specification," Technical Report 92-02-02, Radiation Oncology Dept., Univ. of Washington, Seattle, Mar. 18, 1992.
[21] I. Kalet, "SLIK Programmer's Guide," Technical Report 92-02-01, Radiation Oncology Dept., Univ. of Washington, Seattle, Mar. 17, 1992.
[22] I. Kalet, "Artificial Intelligence Applications in Radiation Therapy," *Advances in Radiation Oncology Physics: Dosimetry, Treatment Planning, and Brachytherapy*, J.A. Purdy, ed., pp. 1,058–1,085, 1992.
[23] Kernighan and Ritchie, *The C Programming Language*. Englewood Cliffs, N.J.: Prentice Hall, 1978.
[24] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object Oriented Programming* vol. 1, no. 3, pp. 26–49, Aug./Sept. 1988.
[25] G.J. Kutcher, R. Mohan, J.S. Laughlin, G. Barest, L. Brewster, C. Chue, C. Berman, and Z. Fuks, "Three Dimensional Radiation Treatment Planning," *Dosimetry in Radiotherapy: Proc. Int'l Symp. on Dosimetry in Radiotherapy*, vol. 2, pp. 39–63, Vienna, Sept., 1988.
[26] M.M. Lehman, "Program Evolution," M.M. Lehman and L.A. Belady, eds., *Program Evolution: Processes of Software Change*, Academic Press, London: Harcourt Brace Jovanovich, pp. 9–38, 1985.
[27] B. Meyer, *Object-Oriented Software Construction*. Cambridge: Prentice Hall, 1988.
[28] D. Notkin, D. Garlan, W.G. Griswold, and K. Sullivan, "Adding Implicit Invocation to Languages: Three Approaches," *Proc. JSSST Int'l Symp. Object Technologies for Advanced Software*, Nov. 1993. Also appears as Springer-Verlag Lecture Notes in Computer Science volume no. 742.
[29] Object Management Group, Inc., *CORBAservices*, Apr. 1995.
[30] W. Paluszynski, *Designing Radiation Therapy for Cancer, an Approach to Knowledge-Based Optimization*, PhD Dissertation, Univ. of Washington, 1990.
[31] D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 5, no. 12, pp. 1,053–1,058, Dec. 1972.
[32] H. Petroski, *Design Paradigms: Case Histories of Error and Judgement in Engineering*. Cambridge: Cambridge Univ. Press, 1994.
[33] S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 4, pp. 57–66, July, 1990.
[34] D. Riehle, "Integrating Implicit Invocation with Object-Orientation through the Event Notification Pattern," *Theory and Practice of Object Systems (TAPOS)*, Special Issue on Patterns, 1996, to appear.
[35] J. Rosenman, G.W. Sherouse, H. Fuchs, S. Pizer, A. Skinner, C. Mosher, K. Novins, and J. Tepper, "Three-dimensional Display Techniques in Radiation Therapy Treatment Planning," *Int'l J. Radiation Oncology, Biology and Physics*, vol. 16, pp. 263–269, 1989.
[36] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Englewood Cliffs, N.J.: Prentice Hall, 1991.
[37] R.W. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Graphics*, vol. 5, no. 2, pp. 79–109, 1986.
[38] D. Garlan and D. Notkin, "Formalizing Design Spaces: Implicit Invocation Mechanisms," *VDM '91, Formal Software Development Methods*. Appears as Springer-Verlag Lecture Notes in Computer Science #551, Nov. 1991.
[39] D. Garlan and M. Shaw, "An Introduction to Software Architecture," V. Ambriola and G. Tortura, eds., *Advances in Software Eng. and Knowledge Eng.*, vol. 1, New Jersey: World Scientific Publishing Company, 1993.
[40] G. Steele, Jr., *COMMON LISP, the Language*, 2nd edition, Burlington, Mass.: Digital Press, 1990.
[41] B. Stroustrup, *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 1986.
[42] K. Sullivan and D. Notkin, "Reconciling Environment Integration and Software Evolution," *ACM Trans. Software Eng. and Methodology*, vol. 1, no. 3, July 1992.
[43] K. Sullivan, *Mediators: Easing the Design and Evolution of Integrated Systems*, PhD Thesis, Technical Report 94-08-01, Dept. of Computer Science and Eng., Univ. of Washington, Seattle, Aug. 1994.
[44] R.N. Taylor, R.W. Selby, M. Young, F.C. Belz, L.A. Clarke, J.C. Wileden, L. Osterweil, and A.L. Wolf, "Foundations for the Arcadia Environment Architecture," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments*, P. Henderson, ed., Boston, Mass., Nov. 28–30, pp. 1–13, 1988.
[45] G. Wiederhold, "Mediation in Information Systems," *Computing Surveys*, vol. 27, no. 2, pp. 265–267, June, 1995.



Kevin J. Sullivan (M'95) received the BA degree in mathematics and computer science from Tufts University in 1987, the MS. degree in computer science and the PhD degree in computer science and engineering from the University of Washington, Seattle, Washington, in 1990 and 1994, respectively. Since then he has been assistant professor in the Department of Computer Science, School of Engineering and Applied Science, at the University of Virginia, Charlottesville, Virginia. His research interests are in software engineering in general, with a focus on software design and integration.



Ira J. Kalet received the BA degree in physics from Cornell University in 1965 and the PhD degree in theoretical physics from Princeton University in 1968. His dissertation was on "Low Energy Theorems for Meson Production by Real and Virtual Photons." He held research and faculty appointments at the University of Washington Physics Department, the Robert Hutchins School of Liberal Studies at Sonoma State College, and the Graduate School of Education at the University of Pennsylvania before coming to the University of Washington in 1978 in the then newly formed Department of Radiation Oncology.

He is currently associate professor in the Department of Radiation Oncology at the University of Washington School of Medicine. He is also adjunct associate professor in the Department of Computer Science and Engineering, the Department of Biological Structure, and the Program in Bioengineering at the University of Washington.

Dr. Kalet's research interests include computer graphic simulation of radiation treatments for cancer, computer control of radiation therapy machines, software development methodology, and artificial intelligence applications to medicine, particularly expert systems. He was the program chair for the Artificial Intelligence in Medicine (AIM) Workshop in 1987.

Dr. Kalet is a member of the Association for Computing Machinery (ACM), the American Association for Artificial Intelligence (AAAI), the IEEE Computer Society, The American Association of Physicists in Medicine (AAPM), and the American Association of Physics Teachers (AAPT).

David Notkin (S'77–M'82–S'83–M'83) is a professor in the Department of Computer Science and Engineering at the University of Washington. Before joining the faculty in 1984, he received his PhD at Carnegie Mellon University in 1984 and his ScB at Brown University in 1977. He received a National Science Foundation Presidential Young Investigator Award in 1988, served as program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, served as program co-chair of the 17th International Conference on Software Engineering, chaired the Steering Committee of the International Conference on Software Engineering (1994–1996), and currently serves as charter associate editor of both *ACM Transations on Software Engineering* and *Methodology and the Journal of Programming Languages*. His research interests are in software engineering, in general, and in software evolution, in particular.