Mediators in a Radiation Treatment Planning Environment *

Kevin J. Sullivan Ira J. Kalet†‡
David Notkin†
Computer Science Department
University of Virginia
Charlottesville, VA 22903 USA
†Department of Computer Science and Engineering
‡Department of Radiation Oncology
University of Washington
Seattle, WA 98195 USA
sullivan@cs.virginia.edu, ira@radonc.washington.edu, notkin@cs.washington.edu

June 25, 1995

Abstract

We describe the architecture of Prism, an integrated system for planning radiation treatments for cancer patients. This architecture is based on the mediator design approach. The problem addressed by this approach is that common methods of designing integrated systems lead to undue module coupling, significantly complicating software development and evolution. The mediator approach was devised to ameliorate the conflict between integration and ease of software development and evolution. It does this by enabling the composition of visible and independently defined components into behaviorally integrated systems. We present Prism as evidence for two claims: first, the mediator method can overcome the problem of coupling, easing the design and evolution of real integrated systems; second, the method profitably can be learned and used by practicing software engineers. Prism continues to evolve in clinical use at University of Washington and Emory University Cancer Centers, and in trials at other institutions, including the University of Miami. It is an attractive subject for continuing research on integration, software structure, and the evolution of mediator-based architectures.

*This work was supported in part by grant number R01 LM04174 from the National Library of Medicine and contract number N01 CM97566 from the National Cancer Institute, by a grant from General Electric Medical Systems, and by the National Science Foundation under Grant Numbers CCR-9113367, CCR-8858804, and CCR-9502029, and by SRA (Tokyo Japan).

1 Introduction

We describe the software architecture of Prism, a system for radiation treatment planning [Kalet et al. 91]. The Prism architecture is based on the mediator approach to designing integrated systems [Sullivan and Notkin 92, Sullivan 94]. The problem addressed by this design method is that building integrated systems using common software design approaches leads to tight coupling of module definitions that significantly complicates software development and evolution. The mediator approach was designed to resolve the tension between behavioral integration and ease of development and evolution by enabling the semantically rich behavioral composition of visible, independent components into integrated systems.

Prism is a real system. It is the radiotherapy planning tool in several large research hospitals, including the University of Washington and Emory University, and is in use in trials elsewhere, including the University of Miami. Prism also continues to evolve, providing an attractive subject for continuing research on integration, software design, and the evolution of mediator-based systems.

We present Prism as strong evidence for two claims: first, to a significant degree, the careful and consistent application of the mediator approach can overcome problems of coupling due to behavioral integration, easing design and evolution; second, this approach can be learned and applied profitably by software engineers in production settings.

This paper is organized as follows. Sections 2 and 3 survey the problem of structural coupling in behaviorally integrated systems and the mediator approach to addressing it. Section 4 introduces radiation treatment planning and motivates the need for integrated systems to support this task. Section 5 introduces key concepts behind the mediator approach through a simple example (a user interface widget viewed as an integrated system). Section 6 discusses how mediators were used in several Prism subsystems. Section 7 presents data on the impact of the mediator approach on system evolution. Section 8 provides data on the size of Prism and of the development effort. Section 9 concludes with an evaluation of Prism as evidential support for our claims.

2 Overview of the Problem

A software system intended to support a complex task, such as radiation treatment planning, should provide a set of effective, easily used, tightly integrated tools. Integration is important both to relieve users from the tedious and error prone task of coordinating tools manually, and to provide rapid feedback on the effects of incremental changes, which is essential to iterative, incremental tasks such as radiation treatment planning.

Unfortunately, integrating the behaviors of tools is often achieved only at the cost of undue complexity in the structure of the underlying software. That complexity increases design difficulties, development costs, and risks and complicates or even inhibits software evolution. Taylor expressed the dilemma nicely.

> "... 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 [Taylor *et al.* 88, p. 2]."

In earlier work, Sullivan and Notkin showed that common design methods unnecessarily complicate integrated system designs [Sullivan and Notkin 92, Sullivan 94]. In particular, design methods that take abstract data types (ADTs) as a basis for modularization do not handle integration requirements very well. Object-oriented extensions, such as classes, inheritance, polymorphism, dynamic binding, and multiple dispatch, do not overcome the basic problems. Nor do common ways of using implicit invocation (event notification).

The problem in a nutshell is that common approaches compromises the independence or visibility of the components whose behaviors are to be integrated, and they often do not isolate aspects of the system responsible for integration. Dependencies complicate design and evolution by preventing components from being conceived, understood, realized, debugged and enhanced in isolation. Hiding components complicates evolution by making it hard to integrate new behaviors with those of the hidden components. Non-modular representations of integration concerns—of what we call behavioral relationships—makes it harder to add, remove, and change these relationships independently.

3 Overview of Our Solution

The mediator method eases the design and evolution of integrated systems by structuring them as collections of visible, independent components integrated in networks of explicitly represented behavioral relationships. The approach combines two basic ideas, applied in two phases. The first phase articulates the decomposition of the system into modules. The second defines phase the module interfaces in a form suitable for efficient implementation.

We call the first phase *behavioral entity-relationship (ER) modeling* because it applies entity-relation *data* modeling ideas [Chen 76] to decompose integrated system *behaviors*. One first identifies the behavioral components of the system that should be independently developed and (re)usable. Then one separately identifies the behavioral relationships needed to integrate these behaviors to produce the required system behavior.

In the second phase, one realizes each behavioral component and relationship as a corresponding object (an instance of a type) in a way that ensures that the two properties hold. First, objects implementing component behaviors must be visible (directly callable and observable) and independent (containing no references to external objects nor code implementing other entities or relationships). Second, behavioral relationships must be represented as separate "mediator" objects.

Unfortunately, ADTs do not directly support this approach. Integrating the behaviors of ADT-based objects requires one of two approaches. Objects can call each other (directly or indirectly), but this compromises independence and also represents behavioral relationships implicitly; or, objects can be encapsulated in "wrappers" that ensure that calls to one are accompanied by required calls to the others, but this compromises visibility.

Nor does adding an event notification mechanism itself solve the problem. There are at least two problems with common uses of event mechanisms. First, the events components announce are often not declared in their interfaces. This makes it harder to reason about components by leaving key abstractions implicit [Reiss 90, Meyers 91]. Second, when objects must interact (e.g., when a change to a model should cause a view to update), one object often registers with and handles events from the other [Goldberg and Robson 83]. This compromises independence and represents the behavioral relationship implicitly.

To begin to address these problems, the mediator method first adopts the abstract behavioral type (ABT) as a basic module construct. An ABT abstractly defines a class of objects in terms both the operations the objects support and the events they announce [Sullivan and Notkin 92, Sullivan 94]. (Other approaches based on a similar idea include TICKLE [Collins et al. 91] and the ISO "MIM" [ISO MIM 91].)

Next, we implement the behavioral components of behavioral ER models as independent, visible ABT-based objects whose events and operations suffice to support their clients.

Finally, we define mediators that integrate these objects by registering with their events and by responding to event announcements from one object by (computing as necessary then) calling operations of the others. In this way, mediators separate behavioral relationships from the objects to be integrated and represent relationships explicitly and abstractly.

Object in multiple behavioral relationships are "monitored" by several mediators. A system as a whole is structured (recursively) as a set of independent, visible objects integrated in a network linked by mediators. Prism demonstrates the success of this method in the hands of a group developing a substantial product. Careful application of the approach led to a radiation treatment planning system with an architecture that is clear, modular and adaptable despite the tight behavioral integration of a broad set of functions.

4 Radiation Treatment Planning Systems

A radiation treatment planning (RTP) system is a collection of software tools used by expert radiation treatment planners, called dosimetrists, to design plans for treating cancer patients with radiation. A treatment must deliver a prescribed dose of radiation to a cancerous target region without causing unacceptable damage to surrounding tissues (e.g., spinal cord). A radiation treatment plan defines a three-dimensional configuration of radiation beams and implanted radiation sources relative to a specific patient that satisfies these constraints.

Designing treatment plans can be hard. Different people have different shapes. The locations and types of tumors vary. Different hospitals have different kinds of treatment equipment. Several kinds of radiation can be used. The space of configurations of radiation sources is vast. The problem is to model this space and search it for good treatments.

A RTP system supports treatment planning with a range of modeling, evaluation, and database tools. Such systems model treatment equipment, patient data, anatomy, kinds and configurations of radiation sources, the radiation fields generated by these sources interacting with anatomy, and so on. These systems also provide tools for visualizing plans and radiation fields, and for managing databases of patients, treatment machines, etc.

4.1 Related Systems

While many RTP systems have been built [Goitein et al. 83, Kutcher 88, Fraass et al. 87, Rosenman et al. 89], including several at the University of Washington [Kalet and Jacky 82, Jacky and Kalet 87b], none before Prism combined the rich functioning, tight integration, and adaptability Kalet saw as necessary for a highly effective treatment planning capability. Previous systems (including Kalet's) exhibited a number of shortcomings.

In many systems, different planning tasks are handled by different, stand-alone, "Unix-like" tools that run as separate processes and are loosely integrated using shared files. Kalet's first system [Kalet and Jacky 82] integrated modeling, dose computation, and visualization tools in this way. To modify the anatomical model, the dosimetrist has to terminate the dose display, run and then terminate anatomy modeling tools, execute the dose computation program, and then restart the dose display program. The awkwardness of such systems increases the cost and diminishes the quality of treatment plans.

Second, the interfaces of earlier systems were inflexible. One uses Kalet's second system [Jacky and Kalet 86, Jacky and Kalet 87b] by constantly traversing a broad and deep menu tree to change plans, anatomical models, visualization parameters, etc. The visualization system is also inflexible, displaying a fixed set of views in a fixed screen layout. These restrictions slow the definition and evaluation of treatment plans.

Third, to the extent that the functioning of earlier systems is rich and integrated, the software tends to be hard to develop and change. Despite its object-oriented structure, Kalet's second system did not accommodate the integration of AI-based planning tools [Kalet 92c, Paluszynski 89a]. Architectural inflexibility is a serious problem because it discourages the integration of rapidly evolving treatment planning capabilities.

4.2 Prism

Prism was developed in an interdisciplinary collaboration between Sullivan and Notkin, software engineering researchers, and Kalet and his colleagues, radiotherapy planning researchers. This collaboration began when Kalet heard a presentation on mediators. With requirements for a demanding new system in hand, but no commitment to a design, Kalet decided to evaluate and eventually to use mediators.

The method seemed an obvious match for the required behavior. It captured the sense of the application as a flexible, dynamic, tightly integrated set of tools, and seemed promising as a reasonably trouble-free approach to design and implementation. In retrospect it indeed appears that it would have been far more difficult, even infeasible, to build a system meeting the requirements with the available resources using common design techniques.

5 The Mediator Approach to Designing Integrated Systems

We detail the mediator method by using it to design a very simple integrated system—a user interface widget called a *dialbox* [Kalet 92]. Three such "subsystems" appear in the Prism *beam panel* (see Figure 1), the tool used to model a radiation beam.

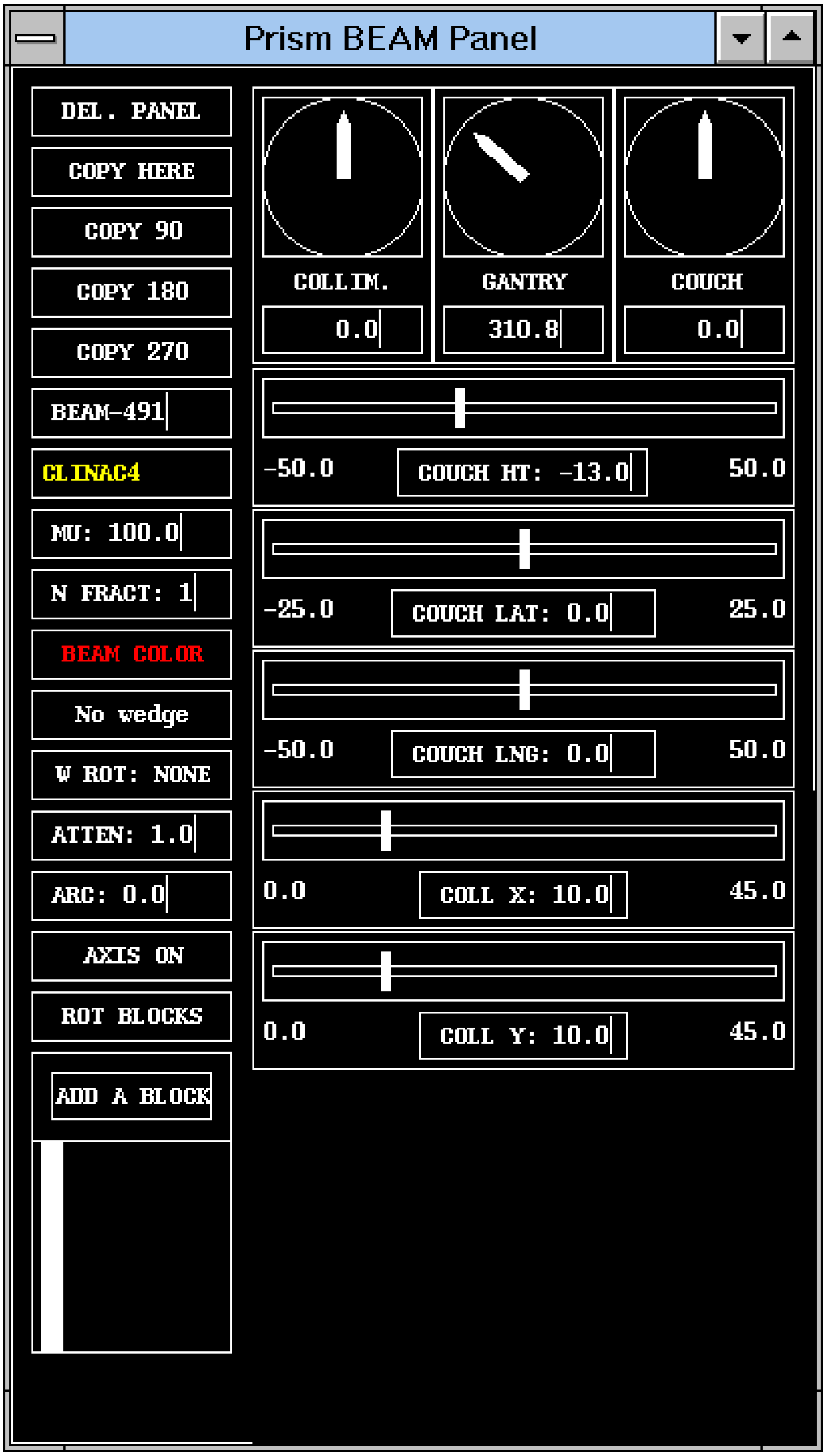

Figure 1: A beam panel with three dialboxes along the top for orienting the beam relative to the patient.

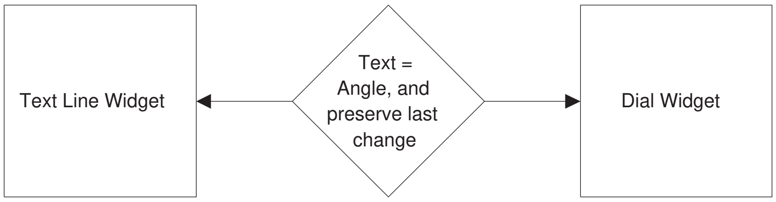

Figure 2: Behavioral ER Model of the behavior of the dialbox widget.

The clock-like widgets along the top of the panel are dialboxes. They are used to orient the beam. Each dialbox has a dial, name, and text line indicating the angle of the dial. The name is set at object instantiation time. The angle is changed dynamically by dragging the dial with the mouse, resulting in calls to an operation of the dialbox object. The text is changed by typing into the text line, resulting in a call to a different operation. The dialbox is an integrated system in that the text line and the dial must present the same value.

5.1 Behavioral ER Model: Maintaining Equality of Scalar Values

When faced with the need for software implementing such a behavior, we begin by designing a behavioral ER model. Figure 2 presents our behavioral ER model for the dialbox behavior. The component behaviors we settled on are *text line* and *dial,* represented informally by rectangles. They store a string and an angle, respectively. The behavioral relationship that integrates them to produce the overall system behavior is represented by the diamond. The diagram as a whole models the dialbox as a composition of visible, independent behaviors.

5.2 Mediator Design using Abstract Behavioral Types

Figure 3 presents a design for the dialbox widget. The boxes represent independent ABT instances implementing the text line and dial behaviors. The left box represents an object exporting operations to get and set the string value of a text line, and an event to signal changes to the string value. The right box depicts a dial ABT with operations to get and set the angle, and an event to signal changes. The objects are visible: their operations can be called and their events monitored directly by other objects. They are also independent.

The diamond represents a *mediator* ABT implementing the behavioral relationship. The mediator works by calling the operations of one object to update its value when notified of a change in the value of the other. If an object calls *SetString(t:Text),* the operation updates the stored value then announces the text line's *StringSet(t:Text)* event. The mediator is notified by invocation of *UponStringSet(t:Text),* having registered this operation with the event (as indicated by the dotted lines in the figure). The operation converts the textual event parameter to a number and calls *SetAngle(d:Degrees)* to update the dial.

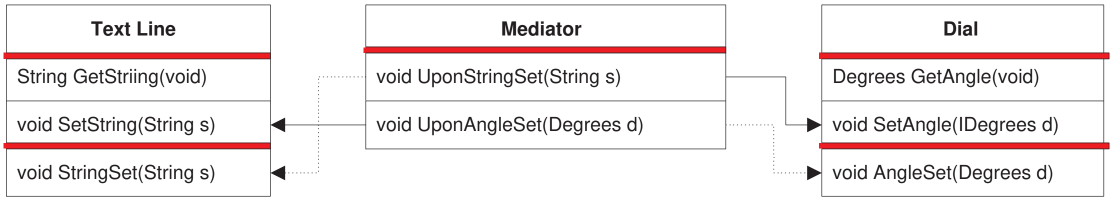

Figure 3: The mediator design of the dial box system.

The change to the dial causes the dial to announce an event. This notifies the mediator again, during processing of the first notification. This could produce a circularity. The circularity can be broken in several ways. One way is for the mediator to maintain a bit to indicate whether an update in already in progress. When notified, the mediator checks the bit. If not set, the mediator sets it, does the update expecting to be re-notified in the process, and finally clears it. When notified during an update, the mediator just returns. We took this approach in Prism [Sullivan and Notkin 92, Sullivan 94].

5.3 Implementing the System in Common Lisp/CLOS

The problem now is to implement the behavioral ER model as a collection of ABT instances. Fortunately, it is straightforward to implement ABTs in many languages. Object-oriented languages are especially well-suited, as they already support objects with operations in their interfaces. The problem then is to represent events in interfaces. Our solution is to represent events as attributes whose values are objects of event types [Notkin et al. 93].

5.3.1 Implementing Events

An event object maintains a set of notifications to be executed when the event is announced and exports operations to manipulate this set and to to announce the event. Figure 4 presents our Common Lisp [Steele 90] implementation of events as a CLOS [Bobrow et al. 88] type. The notification set is an association list that pairs operations to be implicitly invoked with the objects to which they are to be applied. The three event operations are implemented as macros.

- *E.AddNotify(op,ob)* adds the pair consisting of the operation to be invoked *op* and the object (party) *ob* to receive the invocation. In Prism, the receiving party is usually a mediator;
- *E.RemoveNotify(op,ob)* deletes the specified pair from the event's registration set.
- *E.Announce*($p_1, \ldots, p_n$) implicitly invokes all registered objects by iterating overs the (op, ob) pairs, applying each operation *op* (the second entry in the pair) to the associated party *ob* (first entry). For each (op, ob) pair *announce* calls $ob.op(p_1, \ldots, p_n)$,

```
(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)))
```

Figure 4: Common Lisp/CLOS implementation of event objects in Prism.

where p_i are parameters of types as specified by the event's *signature*. In Prism, a reference to the announcing object is always passed as a parameter. (We do not strongly recommend this practice.) The remaining actual parameters are supplied by the announcing object. The operations invoked by the event have to have signatures conforming to the event signature.

5.3.2 The Dial and Text Line ABTs

Implementing the *dial* and *text line* objects now is easy. We just discuss the dial. The key is that the CLOS implementation of a dial has an event attribute and that the dial operations ensure that the event is announced whenever the dial angle is changed. The event announcement is done by a CLOS wrapper ("around") method. Whenever a client invokes *setf* to change the angle, the wrapper is automatically invoked. It erases the dial graphic, does the *setf* to change the angle (within *call-next-method*), draws the new graphic, then announces the *angle-changed* event.

5.3.3 The Dialbox Mediator ABT

Figure 6 presents our implementation of the dialbox mediator (elided to suppress irrelevant details). In this design the mediator serves as the dialbox itself. A dialbox thus aggregates a dial and a text line without hiding them and also integrates their behaviors.

The dialbox type initializer *make-dialbox* creates the text line and dial component parts then registers its own operations with their events. For example, *ev:add-notify* registers the dialbox *db* to be notified by implicit invocation of *upon-angle-changed* when the dial's *angle-changed* event is announced. The parameters to *upon-angle-changed* identify the notified party (the *dialbox*), the announcer (the *dial*), and the new angle. The operation updates the text line using the approach discussed above to handle the circularity, then announces the dialbox's own *angle-changed* event.

```
(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
```

Figure 5: Key features of the implementation of the dial ABT.

That the dialbox announces an event of its own reflects an interesting combination of traits: the dialbox is a mediator from the perspective of the dial and text line, and a more complex, independent object to clients who wish to view it in this more abstract manner.

6 Prism

We appled the simple ideas from the preceding sections to the architectural design of the whole Prism system. The resulting decomposition into independent subsystems linked by mediators provided leverage that substantially contributed to the success of the project. Before discussing the details of mediators in Prism, we describe the main functions of Prism and the information it manages.

6.1 Overview

At runtime, Prism manages a patient case archive, a patient case to be edited, and a set of panels to create, edit and store patient cases. (See Figure 7). These functions are done through *panels,* which are "tools" attached to specific objects, e.g., to organs, tumors, radiation beams. The panel set always contains at least the *patient panel,* which is the Prism "master control."

6.1.1 Patients and the Patient Panel

Figure 8 details the patient case. A case contains patient data (name, number, etc); an optional radiographic image study from which geometric models of the anatomy are built; and sets of organs, tumors, radiation targets, and treatment plans.

```
(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
```

Figure 6: Key features of the implementation of the dial/text line mediator.

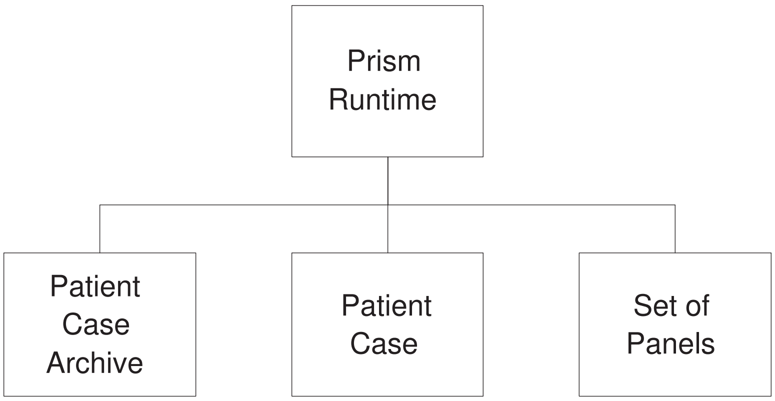

Figure 7: The aggregation structure of the Prism system.

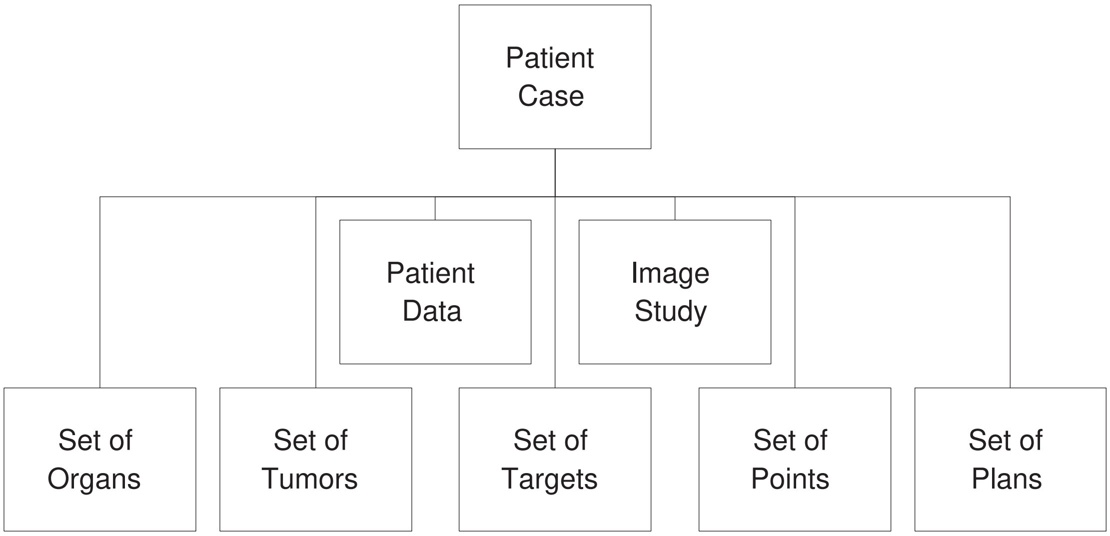

Figure 8: The aggregation structure of a Prism patient case.

Figure 9 illustrates a patient panel with a case loaded. The patient name is Joe Pancreas. Other attributes include the patient number (12345-678), creation date, and comments. Along the side of the panel are buttons to control Prism, e.g., to exit. Along the bottom are menu-like devices called *selectors.* Each selector displays a set of elements (organs, tumors, etc.), and supports launching panels to edit the individual elements.

There are different kinds of panels for editing different kinds of objects: the patient panel for patient cases; the archive panel for accessing the patient archive; easels for editing geometric models of organs, tumors, and the like; plan panels for plans; beam panels for radiation beams; view panels to display graphical views of treatment plans; and block panels for blocks (custom devices for selective attenuation of radiation beams), etc.

6.1.2 Editing Three-Dimensional Objects with The Easel

An easel panel supports input and editing of a three dimensional objects, such as organs. A volume is represented as a sequence of parallel, planar contours, usually constructed by tracing out the object on two-dimensional cross-sectional images (e.g., computed tomography scans). The easel allows one to select from a sequence of images, trace the object of interest, then add the new contour at the appropriate place in sequence of contours.

As shown in Figure 10, an easel has three parts. On the left side is a sub-panel with buttons to terminate the easel and to present attributes such as the color of the contour. Across the top is the "filmstrip." This widget is a scrollable menu for selecting "slices" to edit. Each frame presents a slice of the patient with an image in the background and contours for all modeled objects in the foreground. Finally, the area in the middle of the easel is the actual contour editor.

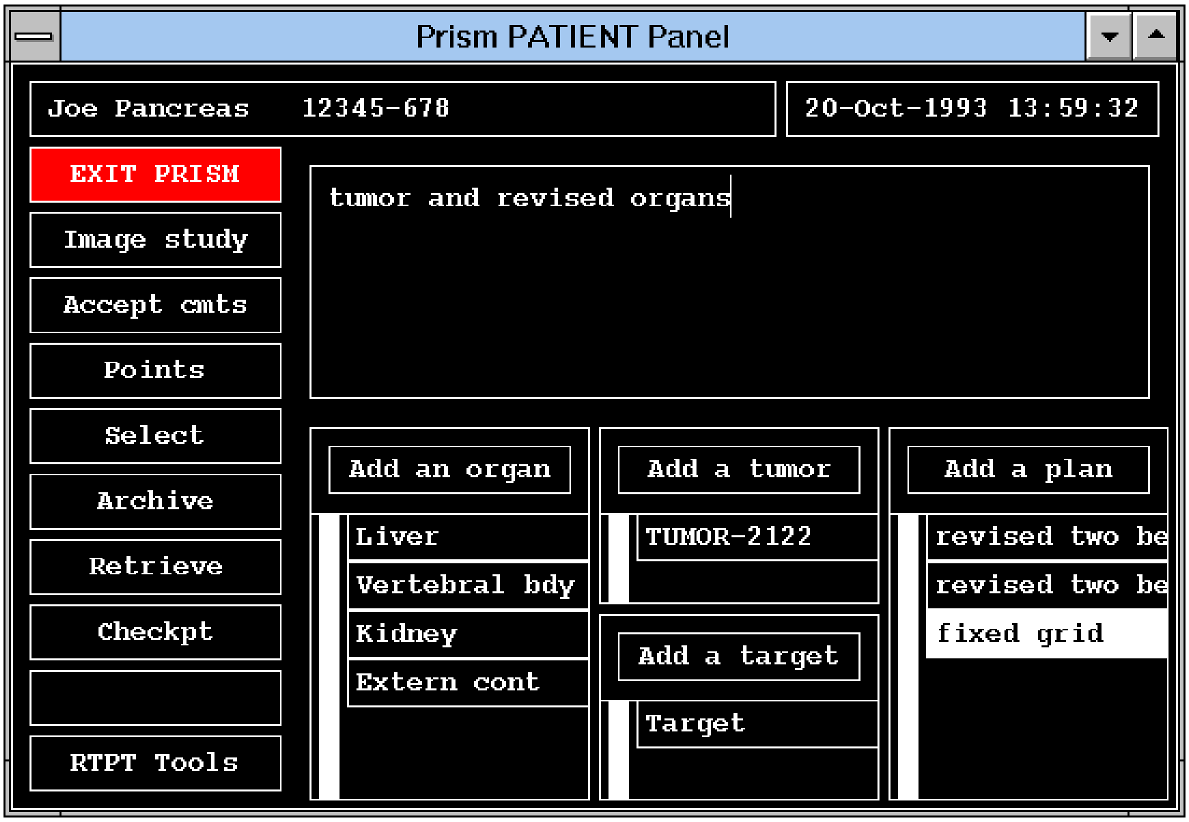

Figure 9: The Prism patient panel.

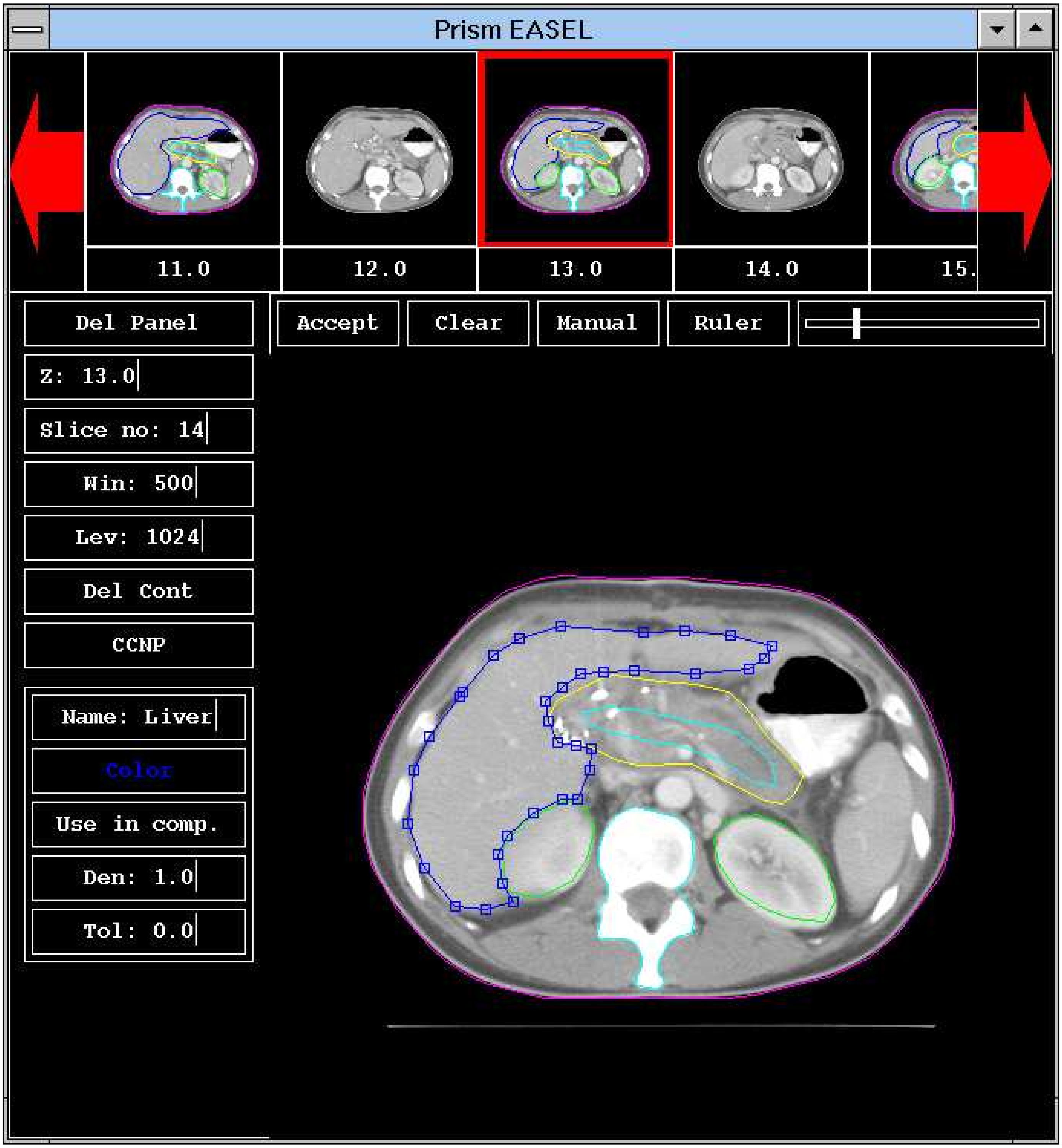

Figure 10: A Prism easel.

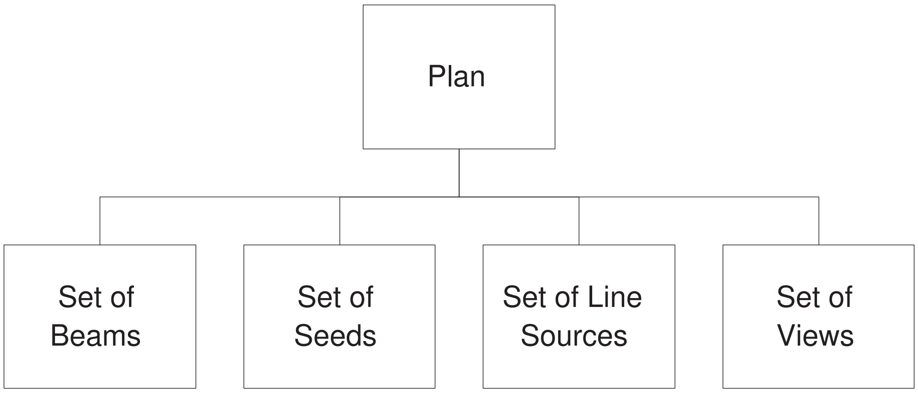

Figure 11: The aggregation structure of a Prism plan object.

One selects a filmstrip frame to bring up a slice on the canvas, edits the contour for the designated structure, and then stores the contour into the model by pressing *Accept*. This action results in all aspects of the system (e.g., computed radiation fields and graphical views) being updated to reflect the new structure. The easel is analogous to the dialbox in design: it is both a mediator (integrating the button panel, filmstrip and contour editor), and an independent, visible object in its own right. A key advantage of this decomposition was reusability of the independent contour editor, as we discuss briefly in Section 7.2.

6.1.3 Plans, Beams & Views

Figure 11 depicts the basic structure of a Prism plan. A plan includes the dosimetrist's initials, plan ID, creation date, comments, and several component sets: radiation beams; radioactive implants (seeds and lines); and graphical views. To edit a plan, one launches a plan panel (see Figure 12) by selecting a plan on the patient panel. In the examples, the plan selected is called "fixed grid." From the plan panel, one can launch additional panels to view and edit plan objects.

Selecting a view brings up a view panel. Figure 13 presents a transverse view, displaying a background image and the two beams in the plan (red and yellow lines). Prism supports both orthographic and perspective views. Perspective views are taken from the origins of radiation beams. The transverse view is one of three orthographic types. Sagittal and coronal views are the other two types, and are perpendicular to transverse views and to each other.

One changes a radiation beam using a beam panel (see Figure 1). As the beam is changed, views are updated. The accessibility and tight integration of tools illustrated by these examples are characteristic of Prism as a whole, as we will now see.

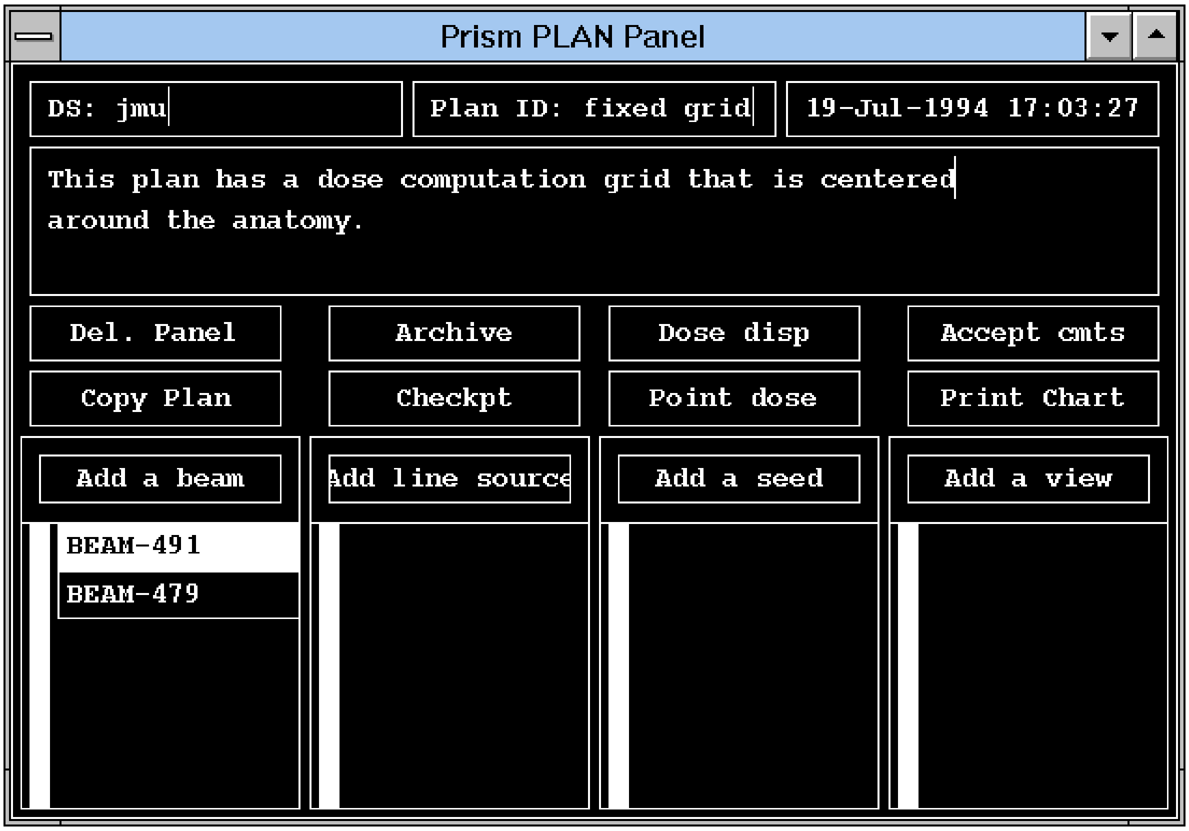

Figure 12: A Prism plan panel.

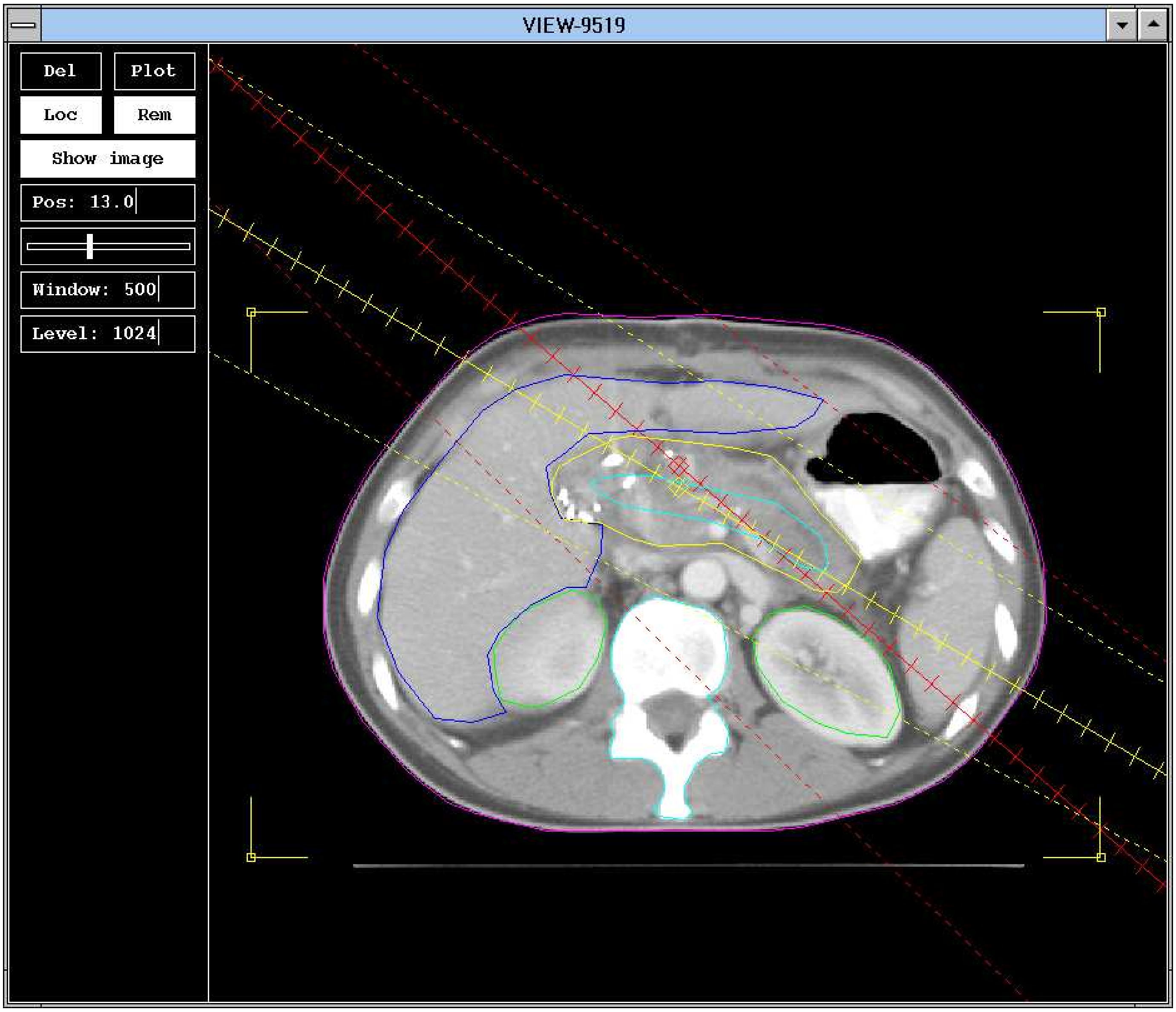

Figure 13: A transverse view of a Prism radiation treatment plan.

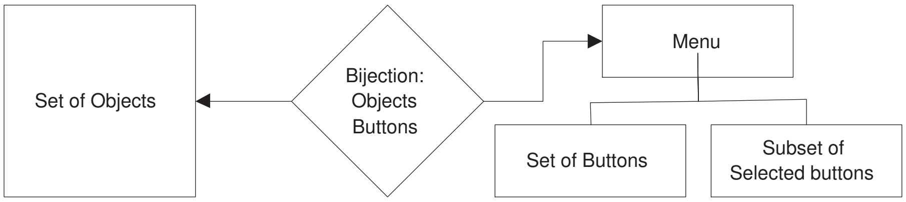

Figure 14: Simplified behavioral ER model of the menu part of selectors in Prism.

6.2 Listing Objects in Selector Menus

We start with selectors. A selector displays the elements of a given set in a menu. On the patient panel for Joe Pancreas (Figure 9), the organ selector indicates that Joe's organs include liver, spine, kidney, and skin. In addition to displaying elements, a selector supports adding and deleting elements, and selecting elements to be edited in separate panels.

6.2.1 Behavioral ER Model: Maintaining a Bijection between Sets

We identified the sets in the patient model and the menus in selectors as independently useful behaviors. The behavioral relationship needed to integrate them to produce the desired overall behavior reflects the requirement that a set and the associated menu remain consistent as either one is changed.

We devised the behavioral ER model presented in Figure 14. The set of objects (e.g., organs) supports element addition, deletion, and iteration. We model the menu as a pair of sets of *buttons:* the set of buttons displayed in the menu, and the subset of selected buttons. The behavioral relationship embodies the requirement that a bijective mapping between objects and buttons be maintained. If an organ is added to the organ set, a corresponding button must be added to the menu. If a button is added to the menu, an organ is added to the organ set. The semantics of sets conjoined with the relationship invariant implies the propagation of behavior needed to satisfy the requirements of the system.

6.2.2 Design: Set ABTs and the Bijection Mediator

We implement the set behaviors as instances of a *Set* ABT with operations to insert, delete, and iterate over elements, and with events to indicate successful element insertion and deletion. The operations announce the events as necessary. For example, *insert(x)* announces *inserted(x)* only if x is actually added (x was not already in the set).

Our 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 collapses the interfaces of the two sets in the abstract model into a single interface. Because the essence of our design involves a bijection between sets, we present the rest of this discussion in terms of sets, ignoring that one is part of a menu.

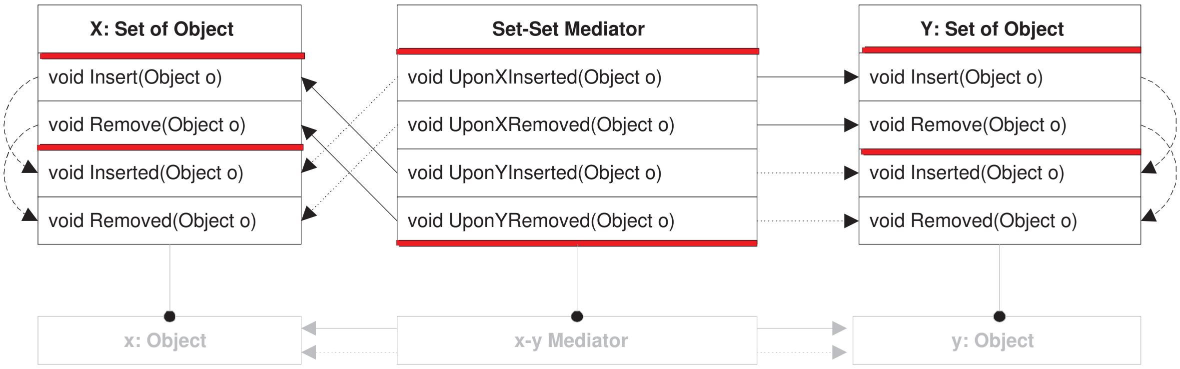

Figure 15: The mediator design of the consistent sets system.

Figure 15 presents our mediator-based design. (We discuss the greyed 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 flow back along the dotted lines. The mediator calls the other set to update it (solid lines), using a "busy bit" to avoid circularities.

Representing multi-valued 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, as illustrated by the mediator that keeps the organ selector menu consistent with the organ set.

6.2.3 Mediators with Complex State

A complexity not discussed above is the way the mediator handles deletions. When an element is deleted from a set, one must find the corresponding element to delete from the other set. To do this, the mediator maintains a relation between buttons and organs, which it updates as the sets change. When an organ is deleted, the mediator looks up the corresponding button, deletes it from the button set, then deletes the association from the relation.

6.2.4 Submediators

The greyed-out boxes in Figure 15 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 (and similarly for all object depicted in all selectors). Thus, if one updated 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.

This "element-wise" consistency is an analog of consistency between dial and text line. We thus define a new mediator type whose instances keep button and element names consistent, and we have the "main" mediator (between sets) deploy and retract these "submediators" as necessary. Thus, when an organ is added to the organ set, the main mediator additionally creates a submediator to keep the organ and button names consistent.

In general, we use this approach to keep corresponding components of structured aggregates consistent in complex ways, without skewing the designs of the components or the aggregates that contain them. The ability to integrate objects while they are dynamically associated in a relation illustrates the leverage provide by separate, explicit representation of behavioral relationships as mediator objects.

6.3 Using Selectors to Dispatch Tool Panels

As noted above, in addition to displaying the contents of sets, selectors also allow selection of objects for viewing and editing in new panels. One could select the plan **fixed grid** for example by pressing the button so labeled on the plan selector on the patient panel. This action highlights the button and launches the plan panel displayed in Figure 12. We now show how we extended the menu system without changing it to implement this behavior.

6.3.1 Behavioral ER Model: Another Bijection

Figure 16 presents the extended behavioral ER model. We have added a panel set and a new relationship (indicated by the coloring in the figure). The relationship requires the panel set to be consistent, in a bijective mapping, with the subset of selected buttons. Selecting a button implies addition of a panel; and closing a panel (and removing it from the panel set) implies deselection of the corresponding button.

The connection from the new relationship to the old indicates the new one "uses" the old. In particular, it uses the relation maintained by the first mediator, between buttons and organs, to determine which organ corresponds to a selected button. This information is needed to associate panels with the organ designated by a selected button.

6.3.2 Design: Reusing Sets and the Bijection Mediator

To accommodate the change in requirements, as reflected in the extension to the behavioral ER model, we just added another set ABT and another bijection mediator the "existing" subsystem. (See Figure 17.) Specifically, we added the set of panels and a mediator between it and the menu. When a button is selected, the new mediator accesses the relation maintained by the first mediator (as depicted by the curving arrow) to find out which element is designated by the selected button. The new mediator then creates a new panel, attaches it to the designated object, and adds the panel to the panel set.

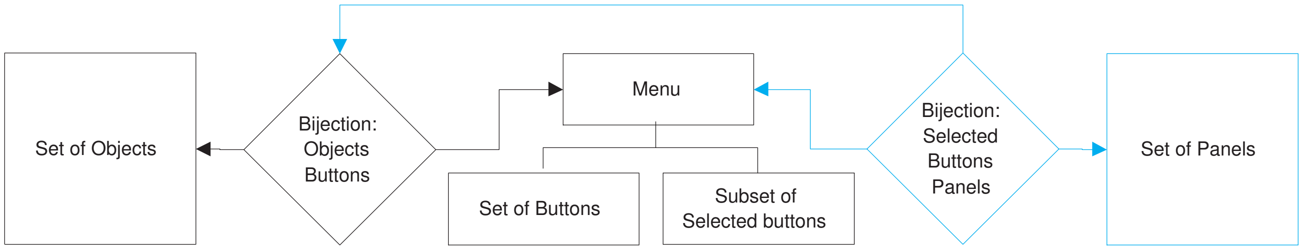

Figure 16: Simplified behavioral ER model of the Prism selector subsystem.

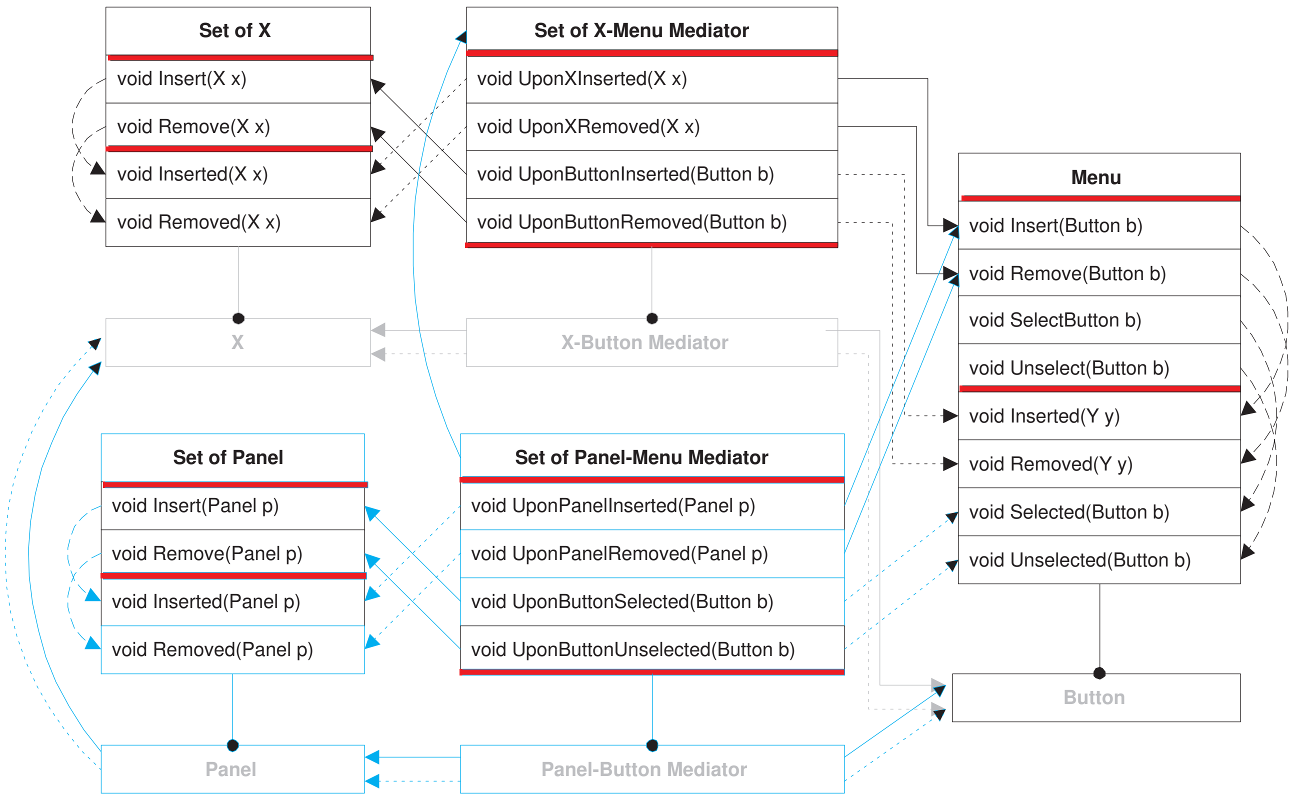

Figure 17: Mediator design of the selector (i.e., tool invocation) subsystem

The extension of the implementation in parallel with that of the behavioral ER model is representative of the way that Prism as a whole was constructed and the way in which it continues to evolve: by behavioral integration of new, independent behaviors, without disruption of the existing design or implementation. With the selector problem solved once, we used the solution for all selectors. The same code, with appropriate parameters, is shared by all selectors. The benefit of mediators here was significant.

This example also illustrates the recursive application of our approach. We viewed the panel set as an independent system to be integrated with the menu subsystem. Then we shifted our focus to the selector as an independent system to be integrated into Prism panels. Panels are then integrated with the patient case, etc. The decomposition of complex behaviors into independent behaviors integrated by mediators at all scales provided much intellectual and managerial leverage. This kind of decomposition meets the goal of modularization: to conquer by effectively dividing [Dijkstra 65].

6.4 Keeping Graphical Views Consistent

Viewing of plans in Prism is supported by view panels. Figure 18 illustrates the four kinds of views that can be displayed in view panels in Prism: orthographic projections along the three axes of the "patient coordinate system," and perspective views taken from the origins of radiation beams. The user can define any number of views (which are stored persistently with plans) 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 through the easel should cause the corresponding graphic in each view to be updated as necessary.

6.4.1 Behavioral ER Model: Maintaining a Cross-Product

Given a set of objects and a set of views, we realized that what we needed was a mediator between the object and view sets; that this mediator would use submediators to keep objects consistent with the corresponding graphics; and that one such submediator would be needed for each element in the *cross product* of the object and view sets. We expressed the basic idea with the behavioral ER model in Figure 19. One of these mediating structures is needed between each object set (e.g., organs, tumors, etc.) and the set of views.

6.4.2 Design: The Cross-Product Mediator

The mediator in this model is similar to that in the selector, but instead of maintaining a bijection, it maintains a cross product relation. As the sets changes, the mediator inserts or deletes elements as necessary, updates its cross product relation, and dispatches

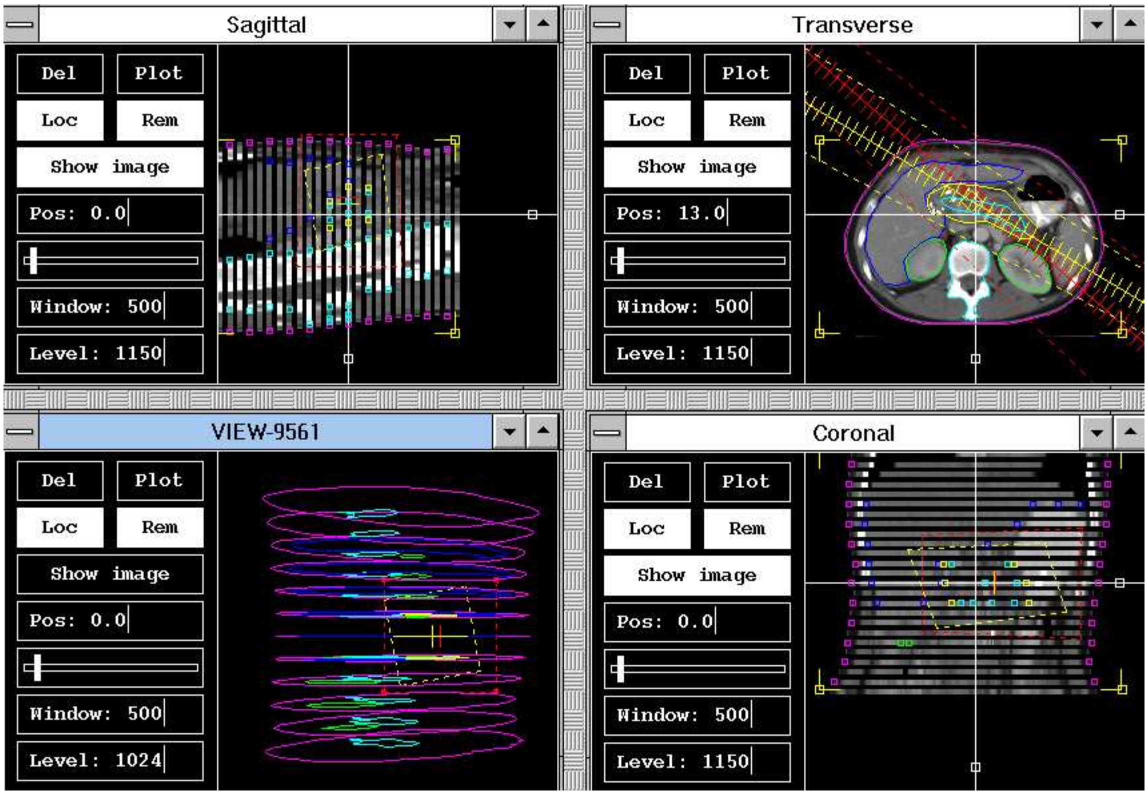

Figure 18: Four view panels: transverse, coronal, sagittal and beam's eye (perspective).

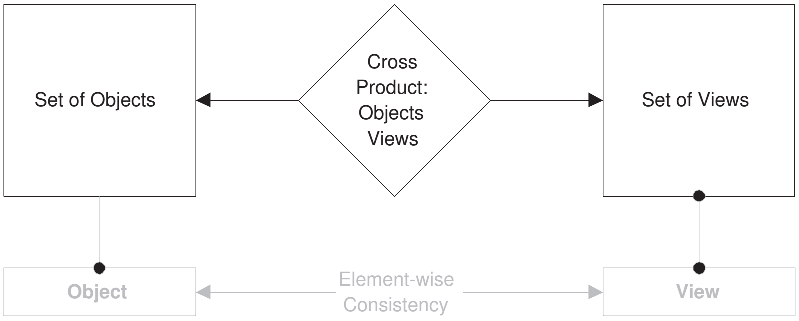

Figure 19: Behavioral ER model for the kernel of the Prism viewing system.

submediators to keep individual objects consistent with their graphical depictions.

The submediators embody design decisions about how to render objects. This design separated the implementation of the graphics pipeline from the of the code that deploys the submediator. We discuss how this modularization eased the evolution of Prism in Section 7.

6.5 Locators Integrate Multiple Views

In this section, we discuss how we enriched the viewing system by having all orthographic views represented in each one. We do this to help the user to combine multiple two-dimensional views into a three-dimensional model. Specifically, each orthographic view presents a set of graphics called *locators,* each indicating how the given view is intersected by the viewing plane of another orthographic view.

A locator is a displayed as a white line. See Figure 18. The vertical locator in the transverse view represents the intersection of the sagittal view. The horizontal locator in the transverse view represents the intersection of the coronal view. The horizontal locator in the coronal view indicates the intersection of the transverse view; and the vertical locator, the intersection of the sagittal view. By rotating a view around a locator, the user can mentally relates the given view to the view depicted by the locator.

Views and locators are integrated. Locators are created and deleted as views are added to and deleted from the view set; and each locator stays consistent with the viewing plane of the view it depicts. So, if one moves the viewing plane of a view, the corresponding locators in other views must be updated. And if one moves a locator (using the mouse), the viewing plane of the other view (and hence also locators depicting that view) must update. Locators thus enable direct manipulation of viewing parameters.

6.5.1 Behavioral ER Model: The Intersects Relation on Views

Behavioral ER modeling led us to the view that, in addition to the set of views, we needed a set of locators, and that these sets needed to be integrated in a relationship modeling the symmetric *intersection relation* on orthographic views. We wanted one locator for each intersection displayed in the intersected view and be consistent with the intersecting view. Figure 20 presents our behavioral ER model.

6.5.2 ABT Design of the View Intersection Mediator

The design for this subsystem is analogous to the designs discussed above. We optimized the actual design by dispensing with the set of locators. The main mediator adds locators directly to the intersected views.

Computing the intersections 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. As these types are encoded in attributes of view objects, only attribute comparison is needed to determine an intersection of a new view with views already in the view set.

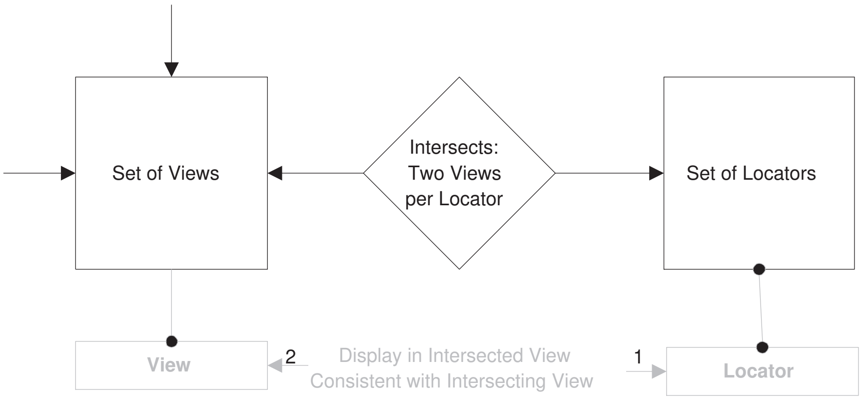

Figure 20: Behavioral ER model of the Prism locators subsystem.

The mediator approach enabled design and implementation of this machinery independently of other roles in which views participate: the relationship to the views selector; the relationships linking views to the elements they display; etc. Mediators preserved the separation of concerns achieved in the behavioral ER models in the design and implementation.

6.6 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 Figure 1) 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, comprising the *collimator subpanel,* shape the collimator opening.

Different machines have different kinds of collimators. Changing the machine (by pressing the button reading *Clinac 4* in the figure to bring up a menu of machine types) may require changing the collimator subpanel. A Clinac-4, for example, requires two slider to adjust its two "jaws." Changing to a collimator with more jaws requires more sliders. Changing to a "multi-leaf collimator" (a kind of collimator we discuss further in Section 7.2) requires a different interface altogether. The behavior we address in this section is the integration of the selection of machine type with the presentation of an appropriate collimator sub-panel.

6.6.1 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 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. The problem is that the specialization dimensions are independent: 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 in contrast with the simplicity of the ideas.

6.6.2 A Mediator Dynamically Selects the Right Part

Instead, we defined a single beam class with attributes to model particle type, collimation system, and other specialization dimensions. We then used inheritance to model different collimator types. 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 sub-panel as an attribute and a hierarchy of sub-panel types.

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 assigned to a beam changes, the beam announces an event. A mediator linking the beam and the beam panel responds by replacing the collimator sub-panel. This ends our discussion of the design of the "core" Prism system against the initial requirements.

7 Evolution of Prism

We applied the mediator design method to requirements that were set before Kalet knew of the mediator method. This is one reason we take Prism as a fair test of the mediator method. We have concluded the method did ease system design. Kalet met demanding requirements on a modest budget. The system is cleanly decomposed into modules. The system was realized in the face of informed opinion that common design approaches would lead to an expensive, poorly modularized and hard to change system.

While strong generalizations are unwarranted, we are confident that Prism provides strong support for the claim that the mediator method can be used to ease the design of a broad class of integrated systems of which Prism is a representative. We are also confident that the key is that the method enables composition of visible, independent components into integrated systems using mediators as separate representations of semantically rich behavioral relationships.

A related question is whether the approach eases software *evolution* [Lehman 84]. Of course, no design method will yield designs that accommodate arbitrary changes with ease. A method encourages the designer to make certain kinds designs decisions, meant by the framer of the method to produce particular properties in resulting designs, such as ease of change in particular dimensions. Ease of change in all dimensions cannot be expected.

Information hiding—the identification and isolation of aspects of systems that tend to change independently—provided the basis for a method that produces systems that are robust with respect to changes in data structures and algorithms [Parnas 72]. The mediator method applies information hiding to different class of changes.

We believe that integrated system tend to change independently in the set of behavioral components they comprise, the individual components, the set of behavioral relationships integrating the components, and the individual relationships. The modularization decisions encouraged and supported by the mediator method ease the evolution of integrated systems by isolating the impacts of changes in these classes.

The claims that integrated systems tend to change in these ways and that the mediator method produces designs that are robust in the face of such changes are rationally appealing but ultimately empirical. While we do not have exhaustive evidence for either claim, we do have two useful data points from Prism that support the claims.

7.1 Corrective Evolution: The Graphics Pipeline

First, recall (Section 6.4) the Kalet isolated the graphics pipeline responsible for maintaining consistency between an object and the graphic that depicts it within the submediators deployed by the "main" mediators that integrate sets of objects with view panels. This modularization paid off when Kalet discovered that his initial design was unworkable. The problem was in incrementally updating contours drawn over background images in the X windows system [Scheifler and Gettys 86]. The behavioral relationship between objects and graphics had to be changed.

The isolation of the graphics code in submediators enabled Kalet to fix the problem with no impact on the superstructure that deployed the submediators or on the objects related by the submediators. We did not have to change or even consider in detail (for instance) the organ set, view set, the mediator between the organ set and view set, or the organs or graphics themselves. Moreover, mediators between the the organ set and other aspects of Prism, such as the menu in the organ selector, were safely ignored. 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.

7.2 Enhancement: Integrating a Leaf Collimator

We conclude the technical discussion of Prism with a brief description of an "unanticipated" change and the ease with which the mediator architecture accommodated it. The change was the integration into Prism of a tool for leaf collimators. A leaf collimator has many adjustable leaves, permitting fine shaping of radiation beams.

The prism leaf collimator comes up when one selects either a CNTS (neutron) or a Philips SL20 beam type, both of which use leaf collimators. On the collimator subpanel of the beam panel, there is a button called leaf display, which when pressed brings up the leaf panel.

The leaf panel shows a beam's eye view, with one textline for each leaf setting. The portal (beam cross section) contour appears. This contour is drawn with the beam portal editor, which is not discussed here, but which reuses the contour editor discussed in the easel section, above. A white, zig-zag outline appears showing the best fit of the collimator leaves to the drawn portal, for the current collimator angle. If the collimator angle is changed (usually done with the dial on the beam panel), the white leaf setting shape changes to show the fit for the new collimator angle, and all the leaf textlines update as well.

The leaf panel was added in the period between clinical implementation (July 94) and November 94. 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 *new-coll-angle* event of the beam object.

8 Development Effort and System Size

To give a sense of the scale of Prism and of the effort that produced it, we sketch relevant 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 radiation fields. Of the Lisp, about 8,300 lines now handle user interface widgets, sets, relations, and events. Prism model objects take 6,200; mediators take 2,200 (excluding code in panels, which include the "object-panel" mediators, optimized into the panel files); panels take 9,500; graphics take 3,500; and the rest includes file input and output and other functions that we count as miscellaneous.

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 take about 60,000 of C [Kernighan and Ritchie] and C++ [Ellis and Stroustrup 90] in GRATIS [Rosenman et al. 89], 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 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 5 person years. Out of this total, requirements specification, done before collaboration began, took 24 person months. Design

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

and implementation, 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, and HP9000 series 700 workstations. 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 for rendering pictures with 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 are performed many times, but the cost in memory and CPU is negligible in comparison with these other functions.

9 Conclusion

Our experience applying the mediator method to Prism has been positive. The architectural benefits of the method played a major role in getting a well integrated highly functional and adaptable system into clinical use. Many factors contributed to this success, including use of a high-level language and avoidance of "gold-plating" in requirements definition; but we are confident in attributing a significant part of the success to the mediator approach.

First, the system is well-structured with respect to accepted criteria: decomposition [Dijkstra 65], information hiding [Parnas 72], modular continuity [Meyer 88], etc. Prism is evidently factored into independent parts integrated by separate mediators representing abstract behavioral relationships; and system evolution has been straightforward.

Second, we can ask the client, Kalet (admittedly a co-author) whether the mediator method helped significantly. Kalet is qualified to answer, having built two earlier systems [Kalet and Jacky 82, Jacky and Kalet 87b] that have seen extensive clinical use, one designed using common object-oriented methods [Jacky and Kalet 87b]. Kalet believes that without mediators, meeting the requirements for Prism would have taken far more resources than were spent or available; and the architecture would have been unnecessarily complex, hard to develop, and inflexible with respect to integration of new capabilities.

Finally, we can ask whether the mediator method helped make Kalet and his colleagues more effective software designers. 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 common object-oriented techniques to using mediators. Superficially, one can believe this is a small shift. Our experience with several members of this project, as well as with students and other colleagues, indicates that the shift is large, requiring judgement and ability developed through coaching and practice.

We believe that designers of integrated systems can benefit in significant ways by the insights developed in this work. To be clear about how we view the epistemic status of our results, we present them as *useful rules of thumb.* In Fred Brooks's lexicon these are "generalizations, even those unsupported by testing over the whole domain of generalization, believed by the investigators willing to attach their names to them [Brooks 88, p.2]." Rules of thumb are distinguished from *truthful and rigorous findings':* "results properly established by soundly-designed experiments and stated in terms of the domain for which generalization is valid [Brooks 88, p.2]." The rationale for presenting well-supported rules of thumb is, finally, instrumental utility: "*Over-generalized findings from other designers' experiences are more apt to be right than the designer's uninformed intuition.* [Brooks 88, p. 2]." In our experience, mediators really work in real systems.

Acknowledgments. Jonathan Unger kindly provided a great deal of technical assistance. Thanks to Matt Conway for pointing out (in his dissertation proposal) the SIGCHI article by Fred Brooks.

References

[Bobrow et al. 88] D.G. Bobrow et al. Common Lisp Object System Specification X3JI3 Document 88-002R. *ACM SIGPLAN Notices 23,* September 1988.

[Brooks 88] F.P. Brooks, "Grasping Reality through Illusion—Interactive Graphics Serving Science," Plenary Address, SIGCHI Bulletin, Conference Proceedings, *Human Factors in Computing Systems,* May, 1988, pp. 1–11.

[Chen 76] P.P. Chen, "The Entity-Relational Model—Toward a Unified View of Data," *ACM Transactions on Database Systems, 1,*1, pp. 9–36, March, 1976.

[Collins et al. 91] T. Collins, K. Ewert, C. Gerety, J. Gustafson, I. Thomas, "TICKLE: Object-Oriented Description and Composition Services for Software Engineering Environments," *Proceedings of the 3rd European Software Engineering Conference*, October 1991, Milan, Italy, pp. 408–423.

[Dijkstra 65] E. Dijkstra, "Programming Considered as a Human Activity," *Proceedings of the 1965 IFIP Congress,* (Amsterdam, The Netherlands: North-Holland), 1965, pp. 213–217, in E.N. Yourdon, Ed., *Classics in Software Engineering,* (New York: Yourdon Press), 1979, pp. 3–9.

[Ellis and Stroustrup 90] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* (Addison-Wesley: Reading, Massachusetts), 1990.

[Fraass et al. 87] B. A. Fraass and D. L. McShan, "3-D Treatment Planning I. Overview of a Clinical Planning System," in I. A. D. Bruinvis, P. H. van der Giessen, H. J. van Kleffens and F. W. Wittkamper, eds., *Proceedings of the Ninth International Conference on the Use of Computers in Radiation Therapy,* (Amsterdam: North-Holland), 1987, pp. 273–277.

[Goitein et al. 83] M. Goitein *et al.*, "Multi-dimensional Treatment Planning: II. Beam's Eye-view, Back Projection, and Projection Through CT Sections," *International Journal of Radiation Oncology Biology and Physics 9,* 1983, pp. 789–797.

[Goldberg and Robson 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* (Reading, Mass: Addison-Wesley), 1983.

[ISO MIM 91] 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.

[Jacky and Kalet 86] Jacky, J.P. and Kalet, I.J., "An Object-Oriented Approach to a Large Scientific Application," *OOPSLA '86 Object Oriented Programming Systems, Languages and Applications Conference Proceedings,* Meyrowitz, N., ed., 1986, pp. 368–376.

[Jacky and Kalet 87b] Jacky, J.P. and Kalet, I.J., "An Object-Oriented Programming Discipline for Standard Pascal," *Communications of the ACM 30,*9, pp. 772–776, September, 1987.

[Kalet and Jacky 82] I. Kalet, and J. Jacky, "A Research-Oriented Treatment Planning Program System," *Computer Programs in Biomedicine 14,* pp. 85–98, 1982.

[Kalet et al. 91] 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 Department, University of Washington, Seattle, WA, October 31, 1991.

[Kalet et al. 92] 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 Department, University of Washington, Seattle, WA, March 18, 1992.

[Kalet 92] I. Kalet, "SLIK Programmer's Guide," Technical Report 92-02-01, Radiation Oncology Department, University of Washington, Seattle, WA, March 17, 1992.

[Kalet 92c] I. Kalet, "Artificial Intelligence Applications in Radiation Therapy," in *Advances in Radiation Oncology Physics: Dosimetry, Treatment Planning, and Brachytherapy,* J.A. Purdy, ed., 1992, pp. 1058–1085.

[Kernighan and Ritchie] Kernighan and Ritchie, *The C Programming Language,* (Englewood Cliffs, N.J., 1978).

[Kutcher 88] 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: Proceedings of an International Symposium on Dosimetry in Radiotherapy 2,* Vienna, September, 1988, pp. 39–63.

[Lehman 84] M.M. Lehman, "Program Evolution," in M.M. Lehman and L.A. Belady, eds., *Program Evolution: Processes of Software Change,* Academic Press, (London: Harcourt Brace Jovanovich), 1985, pp. 9–38.

[Meyer 88] B. Meyer. *Object-Oriented Software Construction,* (Cambridge: Prentice-Hall), 1988.

[Meyers 91] S. Meyers, "Difficulties in Integrating Multiview Development Systems," *IEEE Software,* pp. 49–57, January, 1991.

[Notkin et al. 93] D. Notkin, D. Garlan, W.G. Griswold, and K. Sullivan, "Adding Implicit Invocation to Languages: Three Approaches," *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software* (November 1993). The proceedings appear as a Springer-Verlag Lecture Notes in Computer Science volume.

[Paluszynski 89a] W. Paluszyński, *Designing Radiation Therapy for Cancer, an Approach to Knowledge-Based Optimization,* Ph.D. Dissertation, University of Washington, 1990.

[Parnas 72] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM 5,*12, pp. 1053–58, December, 1972.

[Reiss 90] S. P. Reiss, "Connecting Tools using Message Passing in the Field Environment," *IEEE Software 7,*4, pp. 57–66, July, 1990.

[Rosenman et al. 89] 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", *International Journal of Radiation Oncology, Biology and Physics 16,* 1989, pp. 263–269.

[Scheifler and Gettys 86] R.W. Scheifler and J. Gettys. "The X Window System," *ACM Transactions on Graphics, 5,*2, pp. 79–109, 1986.

[Steele 90] G. Steele, Jr. *COMMON LISP, the Language,* second edition, (Burlington, MA: Digital Press), 1990.

[Sullivan and Notkin 92] K. Sullivan and D. Notkin, "Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methods, 1,* 3, July 1992.

[Sullivan 94] K. Sullivan, *Reconciling Integration and Evolution: Behavioral Entity-Relationship Modeling and Design,* Ph.D. Thesis, University of Washington Department of Computer Science and Engineering Technical Report 94-08-01, August 1994.

[Taylor *et al.* 88] R.N. Taylor, R.W. Selby, M. Young, F.C. Belz, L.A. Clarke, J.C. Wileden, L. Osterweil, A.L. Wolf, "Foundations for the Arcadia Environment Architecture," *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* P. Henderson, Ed., Boston, Massachusetts, November 28–30, 1988, pp. 1–13.