Prism: A Case Study in Behavioral Entity-Relationship Modeling and Design*

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

November 20, 1993

Abstract

We describe the collaboratively developed *Prism*, a tightly integrated but architecturally flexible environment for modeling, visualizing, and simulating radiation treatment plans for cancer patients. Despite significant advances in function, we built Prism at low cost in effort and code size. The key was using *behavioral entity-relationship (ER) modeling* to craft an architecture, and objects representing *abstract behavioral types* (ABTs) to implement it. In more detail, we model a system as a set of independently defined entities used directly by clients but made to work together by behavioral relationships connecting them. We implement this model in an imperative programming framework by representing both entities and relationships as instances of classes that define, announce, and register with events in addition to defining and calling operations. Prism provided a realistic test of the behavioral ER modeling and design method.

*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, and a grant from General Electric Medical Systems, and by the National Science Foundation under Grant Numbers CCR-9113367 and CCR-8858804 and by SRA (Tokyo Japan).

1 Introduction

We discuss a collaboration in which Sullivan and Notkin helped Kalet and colleagues use *behavioral entity-relationship (ER) modeling and design*, a software analysis and design technique [Sullivan and Notkin 92], to build a tightly integrated but architecturally flexible environment for planning radiation treatments for cancer patients [Kalet et al. 91, Kalet et al. 92].

In this collaboration, each of our groups had its own goals—Sullivan and Notkin in software engineering and Kalet and his colleagues in radiation oncology. Kalet had defined the requirements for a radiation treatment planning (RTP) system that was richer and more tightly integrated than previous systems in this domain, and needed a method to help realize the system on a modest budget. Sullivan and Notkin sought to test their claim that their method can ease the development and evolution of sophisticated, tightly integrated environments.

In earlier efforts [McCabe 91, Griswold 91, Sullivan and Notkin 92] Sullivan and Notkin or close associates controlled both the method and the requirements against which the the method was tested. In this case, Kalet had defined the requirements before collaboration began. Nor did Sullivan and Notkin control detailed design or implementation. Rather, Sullivan worked with Kalet on the overall architecture, which Kalet then elaborated and implemented. Thus, in addition to providing a fairer test of the approach, this effort also provided experience in transferring the approach to another group.

The collaboration succeeded. Kalet is now routinely applying behavioral ER modeling and design. Prism implements the original specification with only minor changes. The system is tightly integrated, broad in scope, and has a very flexible architecture. Kalet attributes our success in building this system on a small budget largely to the conceptual and architectural benefits of behavioral ER modeling and design. Despite the anecdotal quality of the evidence, this effort substantially increases our confidence in the ability of this approach to ease development of serious (albeit not immensely complex) integrated environments.

This paper describes both the application and the role of our approach in building it. Section 2 introduces the radiation treatment planning domain, places the Prism system in context with related systems, and details Prism using an example session. Section 3 discusses the software engineering challenges posed by requirements for a combination of tight integration and architectural flexibility, and goes on to present behavioral ER modeling and design as a solution concept. Section 4 shows how we applied the tenets of our approach to design and integrate several key subsystems. We focus on modeling and design idioms that provided leverage needed to make rapid progress in building Prism. Section 5 presents code sizes and the like. Section 6 evaluates this effort and discusses future work.

2 Radiation Treatment Planning

2.1 The Application Domain

An RTP system is a collection of software tools used by dosimetrists to design radiation treatments for cancer. Dosimetrists are expert treatment planners. A treatment delivers a prescribed dose to cancerous target regions without overdosing other organs. A plan basically defines a three-dimensional configuration of radiation beams and other sources relative to a patient that satisfies the given constraints.

Designing treatments can be hard. Challenging cases require care and exploration. The space of possible geometric and anatomical configurations is huge" different people have different shapes; tumor locations and types vary; etc. An RTP system helps by supporting modeling and visualization of plans, computation of dose distributions, visualization of plans and dose distributions, and management of a database of patients, plans, treatment machines, etc. An RTP system that promotes exploratory planning can lead to treatments that would otherwise be missed [Rosenman et al. 89].

A key goal for Prism is to give dosimetrists a powerful exploratory system: one providing a tool set that is broad in scope, dynamically configurable, and tightly integrated. By broad in scope, we mean that the tools support diverse planning tasks: modeling, dose computation, visualization, data management, etc. By flexible, we mean that the dosimetrist can instantiate tools at will, not being constrained to a fixed dialog or layout. Finally, by tightly integrated, we mean that all active tool instances work together interactively and incrementally. When a patient model or plan changes, all graphical renderings displayed by any tool should be updated accordingly, for example.

2.2 Related Systems

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 of these efforts achieved breadth of scope, integration and flexibility. This is not surprising, in light of Taylor *et al.*

> "... 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 88, p. 2]."

This has clearly been the case in the radiation treatment planning domain. First, in many systems, different planning tasks are handled by different, stand-alone, "Unix-like" tools that run as separate processes and are loosely integrated through shared files. Kalet's first system 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.

Second, many existing environments are static and inflexible at runtime. This is often apparent in the user interfaces. One operates in Kalet's second system by traversing a broad and deep menu tree to change plans, update anatomy models, tailor visualization parameters, store models in the database, etc. The visualization subsystem itself is inflexible, too. It displays a fixed set of graphical views in a fixed layout. These limitations make it hard quickly model and simulate patients and treatment plans.

Third, to the extent that existing systems are broad and integrated, their architectures tend to be inflexible and evolution-unfriendly. For example, despite its object-oriented architecture [Jacky and Kalet 86, Jacky and Kalet 87b], Kalet's second system does not easily accommodate integration of prototype AI-based planning tools [Kalet 92c, Paluszynski 89a]. Architectural inflexibility inhibits both research on uses of software in radiation treatment planning and also exploitation of research results by making it hard to integrate new tools—whether prototypes or not—into the overall treatment planning environment.

2.3 Prism

Prism thus serves two purposes: it supports treatment planning in clinical practice, and it provides a platform for research on software technologies in treatment planning. In the second role, Prism provides something like a framework [Johnson 92], to be specialized and extended with additional tools. For example, a tool that computes target treatment volumes based on mathematical models of tumor shape and type, patient movement, etc. is now being devised and integrated. This tool was not foreseen in the original requirements, but was easy to integrate because the Prism architecture is integration- and evolution-friendly.

Both roles—production environment and research platform—demand a combination of tight integration and architectural flexibility. Tight integration of tool instances within a session is needed for efficient treatment planning. Graphical views should provide immediate feedback when models are changed, for example. A flexible software architecture is also needed to support the planning task. In particular, the dosimetrist should be able to instantiate tools and configure the environment at will. The system must support non-deterministic instantiation of tools during a session. Finally, architectural flexibility is needed to accommodate the integration of new tools over the lifetime of the system—as research concepts are tested, refined, and adopted or rejected.

The strength of Prism is that is combines architectural flexibility—in execution and evolution—with tight integration. The simple but powerful anatomy modeling tools can stay on the screen with dose display tools, for example. When the anatomy is updated, the dose distribution tool is easily activated to compute and display the new distribution. To help users understand three-dimensional treatment plans, Prism allows any number of views of a given plan, and a given view can either be displayed or not.

To give a sense of Prism, we discuss a session, using "screen dumps" to help illustrate. It is hard to give a sense for how dynamic Prism is in the written medium. For those wanting a more dynamic experience, the Prism source code is freely available.[1]

[1]For information on obtaining Prism, contact Kalet at an address given on the title page.

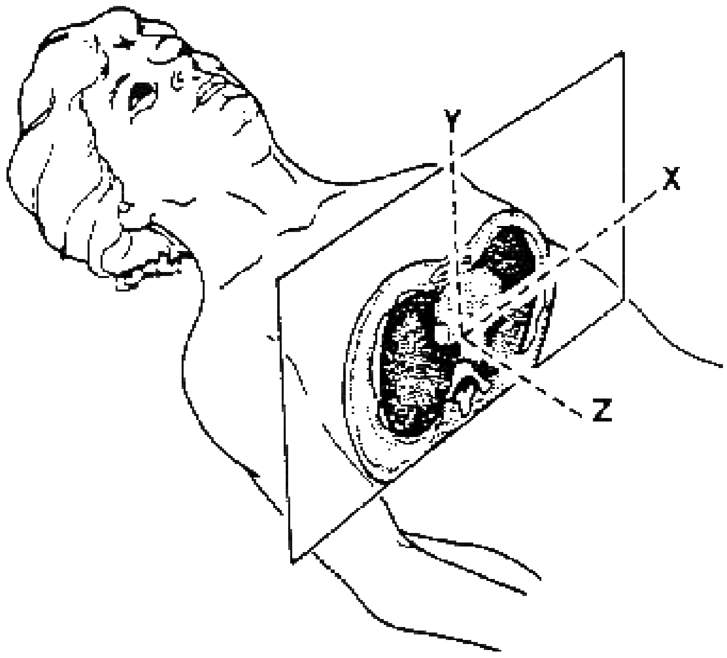

Figure 1: A transverse slice.
This figure illustrates the Cartesian patient coordinate system, and shows a transverse slice through the patient. Images in image studies are oriented in this way.

2.3.1 Image Studies

Image studies are the basis for modeling and visualization in Prism. An image study is a sequence of two-dimensional, radiographic images—usually computed tomography or magnetic resonance scans. Figure 1 illustrates one slice in a study. We build three-dimensional models of the patient anatomy and tumors by tracing contours against successive slices. All contours tagged as belonging to a given organ define the geometric model of that organ. These models, augmented with information such as density and radiation tolerance, are the basis for visualization, dose computation, and other Prism activities.

2.3.2 Master Control: The Patient Panel

Prism tools are presented in *panels.* A *patient panel* appears in the upper left of Figure 2. The patient panel is the tool presented to users on start-up. It is used to select *patient cases* from the database; load associated image studies; edit patient administrative data (e.g., name, hospital ID number), display anatomical models and treatment plans in abbreviated form, invoke tools to display and edit these models, store updates in the database, etc.

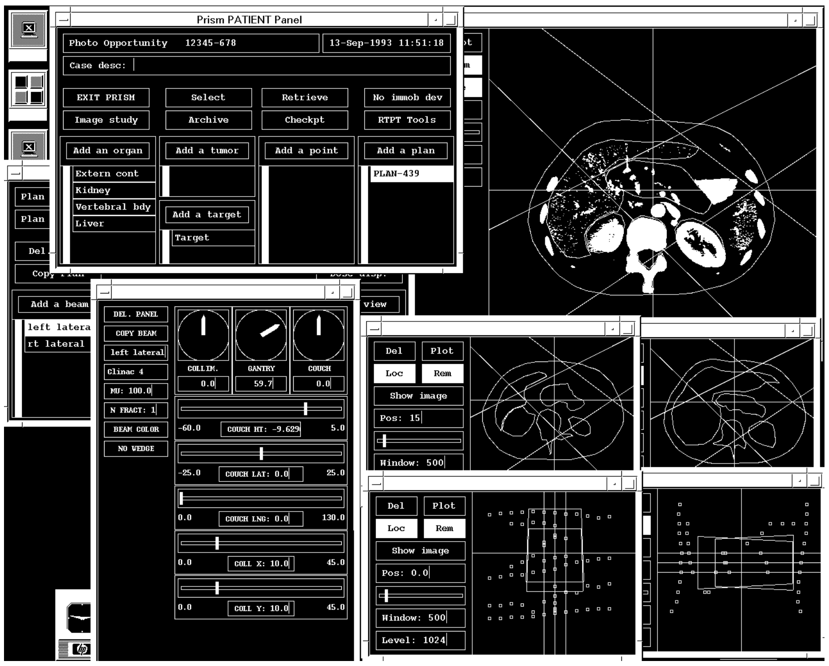

Photo Opportunity 12345-678

13-Sep-1993 11:51:18

Case desc:

MU: 100.0

N FRACT: 1

COLLIM. 0.0 GANTRY 59.7 COUCH 0.0

COUCH HT: -9.629

COUCH LAT: 0.0

COUCH LNG: 0.0

COLL X: 10.0

COLL Y: 10.0

Pos: 15

Window: 500

Pos: 0.0

Window: 500

Level: 1024

Figure 2: A Prism screen.

The bottom part of the patient panel presents several *selectors*. In user interface jargon, these are *multiple selection lists*. Selectors display the patient model in abbreviated form and allow elements to be added, deleted, and selected for further editing in new tool panels. Each selector thus presents a multi-valued attribute of the patient case. There is one for the set of organs defined for a patient, another for the tumors, and so on. When the user loads a patient case from the database, the lists are filled in. Pushing the add button above a selector adds a new element. Selecting an item with the middle button deletes that element. Selecting an element with the left button instantiates a new tool to display and edit the selected object (e.g., organ or plan).

2.3.3 Tool Invocation: The Plan Panel

Selecting a plan in particular instantiates and displays a *plan panel* to display and edit that specific plan. The highlighted PLAN-439 item in the plan selector indicates that that plan has been selected and that there is a corresponding plan panel active on the display. The plan panel for this plan appears in the figure, but is mostly obscured below the patient and beam panels. (The beam panel, discuss below, is the one with the three dials.) Throughout Prism, tool instances are invoked in this way. To edit the radiation target volume named TARGET, for example, one would select the TARGET button in the target selector. This point-and-click tool invocation is what provides the user with a highly flexibly runtime environment.

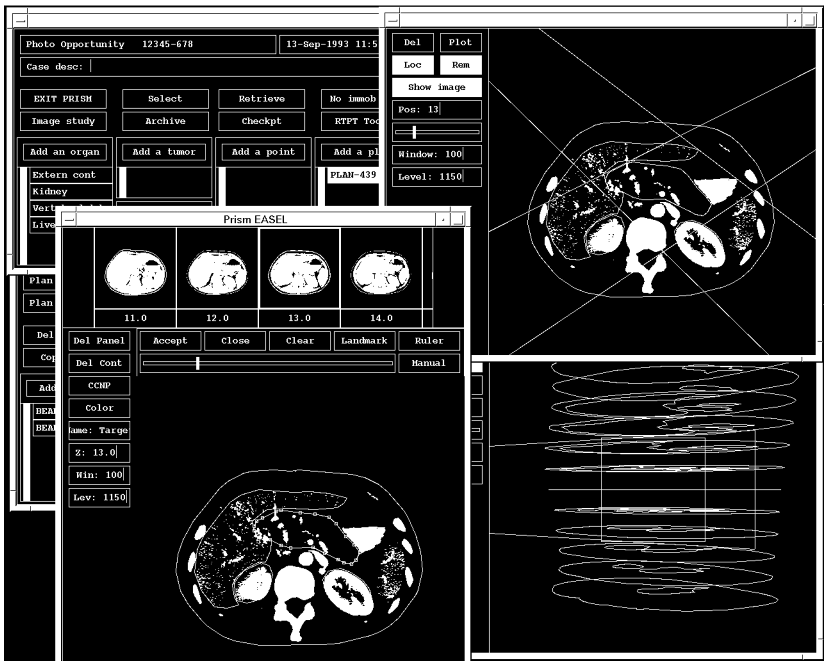
Photo Opportunity 12345-678 13-Sep-1993 11:5

Case desc:

Pos: 13

Window: 100

Level: 1150

Name: Targe

Z: 13.0

Win: 100

Lev: 1150

Figure 3: A second Prism screen.

2.3.4 Physical Modeling: The Easel Panel

The large panel in the lower left of Figure 3 presents the *easel panel* instantiated by the user's selecting this target volume. The easel has several major component parts. Down the left are attributes such as the color in which the contour is displayed, and the Z coordinate of the current slice. Across the top is the *filmstrip.* Each *frame* in the filmstrip presents a slice of the patient model, with the image for that slice in the background and contours for organs, tumors, targets, etc. at that position overlaid. The filmstrip provides a scrollable, visual menu for selecting slices for editing. The large area in the middle of the easel is the canvas, on which editing operations actually take place.

To edit a slice, the user scrolls through the filmstrip to find the desired slice, and then clicks on the desired frame. The third frame is selected here (indicated by highlighting, which is hard-to-see in this window dump). The image and contours for the selected frame are then brought up on the canvas. All contours at that slice are displayed, but only the one for the entity being edited can be changed. In this case it is a contour of the target volume, outlined with small squares to highlight and support editing of this contour. The user can add, delete, and move these vertices. The *Clear* button below the filmstrip deletes the entire contour. In addition to a *Manual* drawing mode, the easel supports automatic contouring based on following edges in the background image. The easel operates on a copy of the contour defined for the underlying patient model. If the user presses the easel's *Accept*, the newly edited contour is inserted into the patient model, replacing the older contour.

2.3.5 Visualization: Views and View Panels

To support visualization of patient cases, including anatomy, tumors, beams, etc., Prism allows the user to define any number of *views* of a given treatment plan. Prism supports orthographic and perspective views. Orthographic views come in three kinds: transverse, coronal, and sagittal. These are perpendicular to the Z, Y, and X axes of the patient coordinate system, respectively. Prism also supports perspective views, from radiation beam origins—"beam's-eye" views.

A selector on each plan panel lists the views for the plan. When adding a view, the user is queried for the kind of view. Views persist even when not displayed. To display a view, the user selects it using the selector. This creates a *view panel,* which displays the view and allows its parameters to be set—position of viewing plane along view axis, whether an image is displayed in the background, etc. A view panel displays all information defined for a plan visible in the view.

Figure 2 presents five orthographic views panels. The large one in the upper right is transverse. It presents the same slice as in the easel in Figure 3, but also displays two radiation beams—as two pairs of diverging lines. One enters from the upper left; the other from upper right. The target volume is in the intersection of the beams, but is imprecisely aimed: the beams hit other organs, including the kidneys (white ovals) and spinal cord (white, irregular shape). This kind of information and the ability to see and change it easily are critical to the dosimetrist.

The two smaller panels below display transverse slices above and below the one in the large panels. The smaller panel in the lower right presents a sagittal view. The squares depict intersections of contours (parallel to the transverse views) with the sagittal viewing plane. The smaller panel to the left displays a coronal view. The overlapping parallelograms depict intersections of the beams with viewing planes. Since beams are shaped as generalized pyramids they intersect with viewing planes in such polygons.

Finally, the lower right panel in Figure 3 presents a beam's-eye view. This view is taken from the perspective of the beam that, in the view above, enters from the upper left. The view depicts a sequence of transverse slices through the patient, as seen from the side. The outer contours represent skin; the inner ones others, organs, target volume, tumor, etc. We anticipate adding *room views*—perspective views taken from arbitrary points in the treatment room—but we have not yet specified or implemented them.

2.3.6 Integration: Easels and Views

The modeling and view tools are tightly integrated. When the user presses *Accept*, new data is inserted into the patient model. This causes all graphics to be incrementally updated to reflect the changes. Even frames in easel film-strips are corrected. If a dose distribution is displayed, and if the change affects the distribution, the data is invalidated and erased from all renderings where it appears. In general, changes made to one part of the environment cause changes throughout, as necessary, to maintain global consistency. It is in this sense that the system is tightly integrated.

2.3.7 Integration: Views with Views

Visualizing three-dimensional configurations from a set of two-dimensional views is hard. Visual hints indicating how views relate to each other can help. Prism provides hints in the form of *locators.* A locator is a line appearing in one orthographic view that depicts the intersection of the viewing plane of some other view. Locators show how views are oriented relative to each other, helping the dosimetrist fuse disparate views into a three-dimensional understanding.

Consider the sagittal view in the lower right of Figure 2. This view depicts a slice that divides the patient into left and right parts. Where the viewing plane intersects the patient is given by the vertical locator in the transverse view, above: the sagittal view is down the middle of the patient. Symmetrically, the horizontal locators in the sagittal view show the relative positions of the three transverse views. The horizontal locator in the transverse views depict the intersection of the coronal view in the lower left. The vertical line in the sagittal view shows the intersection of the coronal view. The horizontal locator in the coronal view presents the intersection of sagittal view.

Views are integrated with each other through locators. The problem is to keep locators and views consistent with each other. The viewing planes of views can be changed, and locators can be "dragged." The position of the viewing plane of a view can be updated using the *Pos* text line on a view panel. If a viewing plane is changed, the locators for that view in other views update. Symmetrically, if a locator is dragged in one view, the viewing plane of the corresponding view is changed. Thus, not only do locators provide hints about relative orientations; they also let the user change relative orientations. The user can animate a viewing plane passing through the patient in one dimension by dragging a corresponding locator in another view. This tight integration significantly eases visualization and exploration of three-dimensional treatment configurations.

2.3.8 Physical Modeling: The Beam Panel

To review, one edits a plan by selecting it using the plan selector on the patient panel. This creates a plan panel. For this discussion, the key features of the plan panel are the two selectors for radiation beams and views. To add a beam, one uses the beam selector. To edit a beam, one selects the corresponding item. That instantiates a beam panel. On the plan panel just visible on the far left of Figure 2, we see that the "left lateral" beam is selected. This instantiated the beam panel, with the dials, in the lower left.

The left side of the beam panel presents widgets for editing the kind of treatment machine producing the beam (Clinac 4), the color in which the beam is displayed in views, etc. Across the top are dials used to adjust the beam orientation relative to the patient. One models rotation of the couch the patient lies on; another models the position of the gantry that supports the beam apparatus, rotating on an axis parallel to the floor; the third models the axial rotation of the collimator through which the beam passes on its way to the patient. The bottom part of the panel displays several sliders. Three adjust the lateral and longitudinal positions and the height of the couch. The rest adjust the collimator aperture.

2.3.9 Integration: Beam Panels with Beams and Views

Beam panels are integrated with views in the sense that changes to a beam are reflected in all views that display the beam. As the couch dial is dragged around, for instance, the pair of nearly parallel lines in the transverse views rotate, and the depiction of the patient in the beam's-eye view seems to rotate in three dimensional space. Changes in most beam parameters also invalidate any dose distribution that has been computed, causing views to erase depictions of out-of-date values.

An interesting aspect of the beam panel is in the integration of the panel with the beam it displays. A user can change the *machine* attribute of a beam (set to "Clinac 4" in this example). Changing the machine may change the collimator used to shape the beam, since different machines have different collimation systems. This, in turn, may require a change in the collimator part of the beam panel, since different collimation systems have different capabilities, hence interfaces. With a Clinac 4, two sliders are required, since there are two "jaws" that can be manipulated. Other devices have more degrees of freedom, and so need an interface with more sliders. With a "multi-leaf collimator," a slider-based interface is not sufficient; a contour editor for specifying the desired beam cross-section is more appropriate. Thus, when the user changes the machine attribute, the user interface presented by the beam panel may itself change, to allow editing of a beam shaped by a different kind of collimator.

3 Behavioral ER Modeling and Design in Prism

The Prism functional requirements call for a system dynamically configured by the user, broad in scope, tightly integrated, and easy to evolve. This presented an interesting, difficult software engineering problem: it was unclear how to meet these requirements on a limited budget. It was easy to imagine an architecture where each tool type would be defined to manage its interactions with a dynamically changing set of other tool instances; but the tangled structure that would result would be costly to build, debug, and extend. It was evident to Kalet that the requirements could not be met within budgetary and time constraints using common (e.g., object-oriented) methods. The problem was to obtain the required tight integration without losing architectural flexibility—the key to controlling development costs and easing evolution.

The behavioral entity-relationship modeling and design approach developed by Sullivan and Notkin [Sullivan and Notkin 92] purported to reconcile integration with architectural flexibility, and with ease of software development and evolution. Kalet found the method sufficiently appealing that he decided to adopt it for the Prism project.

Our method involves an approach to analysis, design, and implementation. In this section we present the method in a nutshell, and illustrate it by showing how we analyzed, designed, and implemented a new interface widget as a simple "integrated environment" in which two simpler widgets work together. The widget we discuss is a *dialbox*, defined in the SLIK user interface toolkit, used to build all Prism interfaces [Kalet 92].

3.1 Our Method

Our method is based on the notion that the conflict between integration and architectural flexibility is reconciled by viewing and implementing systems as collection of visible, independent entities integrated in a network of separate behavioral relationships. Entities provide the behaviors used by system clients—e.g., a dial on an interface, a set of organs in a model of a patient. Behavioral relationships make the entities work together as they are manipulated by clients. A simple behavioral relationship might assert that when a dial is moved, a text representation of its angle is updated accordingly and, similarly, that when the text is changed, the dial updates.

The approach provides a clean model of integration. To make given tools work together, we identify the behavioral relationship we want between them, and implement the relationship as a separate component. This also leads to a flexible architecture. When tool instances are created at run time, we integrate them with other instances by instantiating behavioral relationships to connect them together. We can make a stand-alone dial work together with a text display by creating a relationship object that makes them work together. When a new tool *type* is defined, we integrate it with existing tools by defining new behavioral relationship types. In many cases, existing types need not change.

To provide the essential details, we sum up our analysis, design, and implementation methods in a few lines each. Then we illustrate their use for the simple case of the dialbox widget. We refer interested readers to more "theoretical" discussions elsewhere [Sullivan and Notkin 92, Sullivan 94].

3.1.1 Analysis

- First, identify and represent key entities as visible, independent objects. An object is visible if it can be directly referenced and used by any client. It is independent if it is not defined to reference any objects outside of itself.
- Second, identify and represent behavioral relationships as separate objects. We use the term *mediator* to refer to an object that represents a behavioral relationship. Mediators are dependent—defined to reference the objects they integrate.

3.1.2 Design

- First, design objects as instances of *abstract behavioral types.* An ABT defines a class of objects in terms of an abstract state space, applicable operations, visible activities, and responses to the activities of other referenced objects. The distinction between this and an abstract *data* type is that one ABT-based object can extend the behavior of another by responding to its visible activities.
- Second, design behavioral relationships as objects (mediators) that reference the objects they relate and that integrate them by responding to their activities, calling their operations, and storing locally any additional information needed to maintain the specified relationships.

3.1.3 Implementation

- Implement abstract behavioral types as classes in object-oriented programming languages. Implement abstract state in terms of hidden instance variables. Implement applicable operations as public methods. Implement visible activities as public instance variables whose values are *event objects* (see below). Implement responses to activities of other objects as registrations of hidden methods with the event-valued instance variables of the other objects.

- Implement event objects as instances of event classes supporting three operations. If E is an event object, then $E.Register(op, ob)$ should cause E to store the association between the operation *op* and the object *ob*. $E.Unregister(op, ob)$ should break this association. $E.Announce(p_1, \ldots, p_n)$ should iterate over all registered *op*, *ob* pairs, applying the registered operations to the corresponding objects: i.e., calling $ob.op(p_1, \ldots, p_n)$. The p_i are the parameters that define the *signatures* of both the events and the methods that they "implicitly" invoke.

3.2 Example: The DialBox Widget

To illustrate these ideas, we discuss how we applied them to model, design, and implement a user interface widget called a *dialbox.* The dials in the beam panel (see Figure 2) are instances of this kind of "tool," which can, itself, be seen as a tiny integrated environment.

The key components—the tools—of a dialbox are a *dial* and a *text line*. The angle of the dial can be changed by "dragging" the dial with the mouse. The text in the text line can be changed by typing a new value in the window. Both components also have "API's"[2], program callable interfaces for getting and setting their values. These tools are integrated in that as either changes the other updates so both continue to display the dial angle. As the dial turns, the text line updates repeatedly until the dial's motion stops. When the user types a new value to the text line and presses *Enter,* the dial snaps to the given angle.

It is sometimes desirable to treat it not as a system of related components but as a unity—a single object. A dialbox is instantiated in entirety, for example: one need not create then glue together several components. To support the view of the widget as a unity, we specify that the dialbox as an object has its own "API" with operations to get and set the (common) angle of the underlying dial and text line components.

3.2.1 Analysis

When given requirements for a "complex" behavior, such as the dialbox, we begin an analysis by asking if we can view it as a system of independent behaviors integrated in a network of behavioral relationships. We have no mechanical procedure for finding a "good" decomposition. Our approach is heuristic. We are guided, however, by concerns for issues of independence, relationship, and visibility. What pieces can we use, develop, understand

[2]This stands for *applications programming interface,* or some small variant of this phrase.

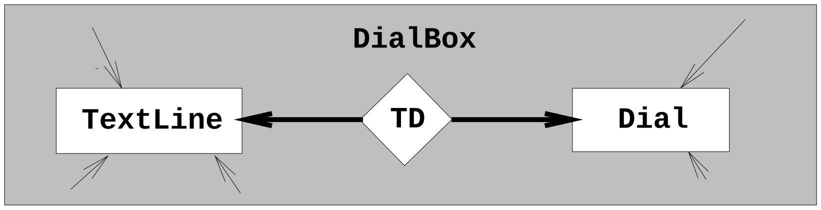

Figure 4: The architecture of the dial box "integrated environment."
The rectangles represent independent entities; the diamond represents the behavioral relationship that integrates them. The dark arrows denote the dependence of the relationship on the entities. The light arrows suggest the entities are visible: used directly by clients.

independently? Given a breakdown into parts, is there a simple statement of the behavioral relationship needed to integrate the parts? What visible structure do the clients of a system "want" to see?

Reasonable answers for the dialbox are fairly obvious. We break it into two independent entities linked by one behavioral relationship, as pictured in Figure 4. The entities are a *dial* and a *text line.* The relationship is simple: if the value of one changes, the other must be updated to reflect the change. This breaks the system into two entities that we can implement, test, and (re)use independently, plus a relationship that is easy to comprehend as a unit in its own right.

3.2.2 Design

At the design level the task is to represent the entities and relationships as ABT instances while preserving the independence and visibility structure of the analysis model. Figure 5 presents the essential features of our design. The text line and dial are defined as independent ABTs. The behavioral relationship is represented as an ABT (a mediator) that responds to the activities of each entity by updating the other to maintains the desired relationship in the face of client accesses.

In more detail, the dial and text line ABTs define operations to *Get* and *Set* their values and events that are announced when these values change—*NewInfo* and *NewAngle*. The mediator is dependent, referencing the dial and text line. It integrates these entities by responding to their event announcements and calling their operations.

Specifically, the mediator registers its *UponNewInfo(x)* operation with the text line *NewInfo(x)* event. When the event is announced, the operation is invoked. It computes a new angle *a* for the dial by converting the updated text—which is passed as an event parameter–to an angle, and then it calls *Dial.Set(a)* to update the dial with the new angle. The behavior is symmetrical for changes to the dial. The circularity that results—one update causes another causes another etc.—can be broken in several ways. One is for the mediator to maintain a bit indicating whether an update is in progress. When invoked, the mediator checks the bit. If not set, the mediator sets it and performs the update. If set, the mediator just returns [Sullivan and Notkin 92].

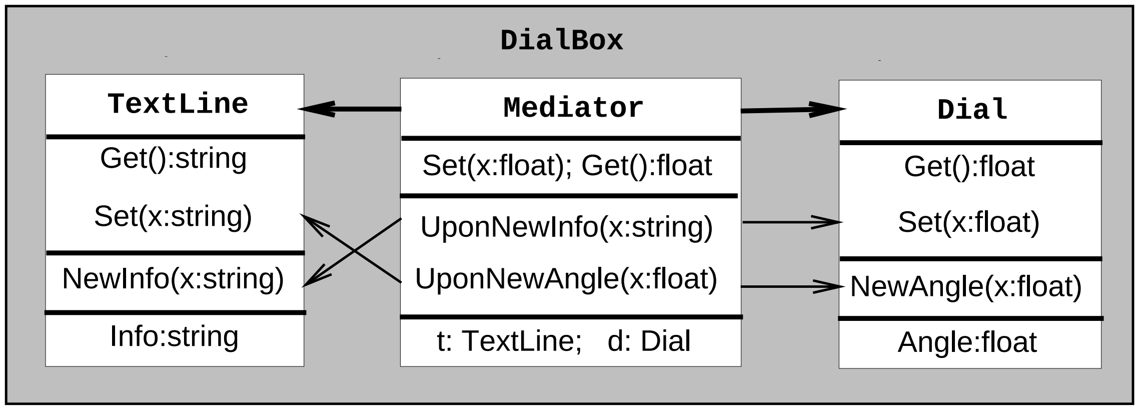

Figure 5: Mediator-based design of the dialbox widget.
The heavy lines indicate dependencies between objects. The dial and text line are independent. The mediator depends on both. The dashed arrows represent registrations of the operations at the tails of the arrows with the events at the heads. The lighter solid arrows represent invocations of the operations at the heads by the operations at the tails.

3.2.3 Implementation

It is straightforward to implement ABTs in common programming languages. The main problem is representing events. This is not hard [Notkin et al. 93]. It is especially easy in object-oriented languages, which already support objects with states and operations. In this case, we represent events in object interfaces as instance variables holding event objects. An event object maintains an association between the event it represents and the objects and operations to be invoked when the event is announced [Sullivan and Notkin 92].

Events. The implementation environment for Prism is Common Lisp [Steele 90] and CLOS [Bobrow et al. 88]. Our first task was to support objects having events as well as operations in their interfaces. We used event object-valued instance variables for this. An event object is an instance of an event class with operations to *Register* and *Unregister* objects and operations and another operation to *Announce* the event.

The mechanism is simple. Source code appears in Figure 6. An event object maintains an association list that records object/operation pairs. Upon creation, the list is empty. The operations, implemented as macros, are simple. *Add-notify* registers an object (*party*) and an operation to be applied to the party. Registration removes any operation already present for the party, then stores the new operation. *Remove-notify* removes any registration for a party. Finally, *Announce* iterates over the list, applying the operation part of each entry to the associated party. The parameters *object* and *args* are passed to the invoked operations. *Object* identifies the object announcing the event. *Args* encodes other parameters for the event. Thus, when a text line announces *NewInfo(x)* by calling the announce operation of this event object, *object* is a reference to the text line and *args* is the new text string x.

```
(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 6: Common Lisp/CLOS implementation of event objects in Prism.

Entities. Implementing the *dial* and *text line* object specified in Figure 5 is now easy. Since the entities are similar, we just discuss one. The dial stores a numerical representation of an angle, defines operations to get and set this value, and makes changes visible by announcing an *angle-changed* event. The problem is in representing and announcing the event. Figure 7 presents the dial class declaration and shows how we guarantee that the event is announced when *setf* is used to update the angle attribute. The idea is to use CLOS wrapper methods. The class includes a slot holding an event object. The wrapper, which is automatically executed whenever *setf* is called, announces the event. (In addition, the wrapper updates the dial graphics; we do not discuss graphics any further).

Mediators. Finally, implementing the mediator is easy. Elided source code appears in Figure 8. In our implementation, the mediator is the dialbox itself. As defined by the *defclass,* a dialbox has a dial and a text line as its visible parts. The *setf* in the dialbox class initialization routine *make-dialbox* creates these objects and assigns them to their slots in the dialbox object. The initialization routine also registers the mediator operations with the events of the dial and text line objects. The first, *upon-angle-changed* handles changes to the dial; the second, *upon-info-changed* handled updates to the text. In the registration instructions (*ev:add-notify*) the variable *db* refers to the dialbox—the party to be notified. The next expressions reference the event objects in the dial and text line objects. The *NewAngle* event is represented by an event object in the dial's *angle-changed* slot, in turn is found through the *the-dial* slot of the dialbox *db.* The parameters preceded by hash signs (#) are the operations to be invoked when the events are announced.

These "update" operations are similar so we just present one. *Upon-angle-changed* updates the text line when the dial changes. The parameters passed to this operation identify the notified party (*dialbox*), the event announcer (*dial*), and the new angle. The operation uses a busy bit to prevent circularity, as discussed above. Between setting and clearing *busy*, the operation updates the string value of the text line, then announces the dialbox *angle-changed* event. This event supports clients that treat the dialbox as a unit.

```
(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 7: 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
```

Figure 8: Key features of the implementation of the dial/text line mediator.

Now consider what happens when a client changes the dial angle using *setf.* The wrapper method is invoked by Common Lisp before *setf* executes. The dial graphic is erased. The *setf* occurs within *call-next-method*, the graphic is redrawn, and then the angle-changed event is announced. This invokes the mediator *upon-angle-changed* operation. This routine checks the busy bit, finds no update in progress, converts the new angle to a string and sets the value of the text line. This causes the text line's *new-info* event to be announced, which invokes the mediator's *upon-info-changed* operation. This operation checks the busy bit but finds an update in progress. Control returns to the text line (its event announcement returns). The text line update completes and returns to the mediator. Now the mediator announces its *angle-changed* event. Finally, the mediator clears the busy state and returns. The original *setf* operation on the visible dial object completes with the whole dialbox system in a consistent state.

4 Modeling and Design Problems and Solution

The Prism architecture is based throughout on these analysis, design and implementation methods. In this section, we discuss five Prism subsystems to show how we applied the tenets of our approach to realize the required integrated Prism functions while preserving architectural flexibility.

We start with multiple selection lists—the interface widgets that display lists of organs, tumors, and plans on the patient panel, beams and views on the plan panel, and so forth. Next, we discuss how these widgets are used in *selectors.* A selector uses a selection list to display a set of objects, such as organs, to support addition and deletion of elements, and to support instantiation of tools to edit elements. Third, we discuss *locators,* the lines displayed in view panels to depict intersections between views. Fourth, we present our architecture for keeping graphical renderings—views, easels, film strips, etc.—consistent as underlying entities, such as beams and organs, are changed. Finally, we discuss our model for changing the interface presented by the beam panel as the beam type is changed.

4.1 Multiple Selection List

A multiple selection list (hereafter *menu*) displays a list of items. Items can be added to and deleted from the menu, and they can be selected and deselected. When selected, an item is highlighted; when deselected, it is unhighlighted.

4.1.1 Analysis

We based our analysis of the menu behavior on knowing how we would use menus in selectors. The idea for selectors—discussed next section—is to have a menu participate in two relationships. The first is a one-to-one correspondence between an object set (e.g., the set of organs of a patient) and all the items in the menu. The second is a one-to-one correspondence between selected menu items and active tool instances. This way, if an organ is added to the patient, an item must appear in the list; and if an item representing

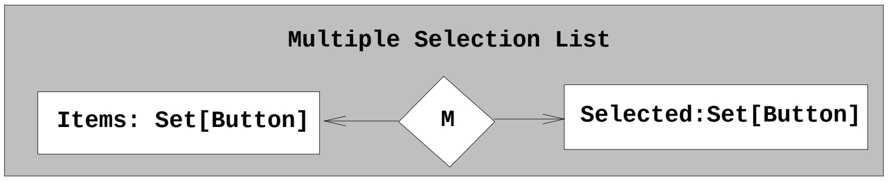

Figure 9: Simplified model of a multiple selection list.
A multiple selection list is viewed as a pair of sets of buttons. A button is an item displayed by a menu that can be selected or deselected. The set of buttons called *Items* contains all buttons displayed by the menu. The set, *Selected,* contains exactly the subset of selected buttons. The behavioral relationship M will ensure that this constraint is maintained as buttons are selected and deselected, inserted and deleted.

an organ is selected, a tool must be instantiated for editing that organ. This implements the behavior specified in the Prism requirements.

Our analysis of the menu itself resulted in the model illustrated in Figure 9. We view a menu as maintaining two sets of *buttons.* The sets are entities into which one can insert and from which one can delete references to buttons. The buttons themselves are entities that can be selected or deselected. The typical usage is for a client to insert unselected buttons into *Items,* then to select and unselect them, and finally to delete them from *Items.*

Integration of these sets and the buttons they contain is achieved by the behavioral relationship (M in the figure). This relationship imposes the constraint that the selected set is the subset of *Items* containing exactly those buttons that are selected. Thus, if a button in *items* becomes selected—because a user invokes its *Select* operation—it must then be inserted into the *Selected* set. The relationship makes the entities work together. Although our implementation does not precisely have this structure, this analysis significantly eased the design, implementation, and integration of the menu components.

4.1.2 Design

In design and implementation, we merged the two sets and the behavioral relationship into a single object. In design and implementation, the menu supports operations to insert, delete, select, and deselect buttons. The select and deselect operations are the equivalents of insertion into and deletion from the selected subset. The menu also announces events for button insertion, deletion, selection, and deselection.

We designed buttons as ABTs with operations for selection and deselection, and events to indicate these activities. The key to the menu design is that when a button is inserted into the menu, the menu registers with the button to be notified of its selection and deselection events. When a button in the menu is selected by the user, the menu is notified. In turn it announces its own *selected* event. This mimics the button being inserted into the selected subset. Deselection is symmetrical. When a button is deleted from the menu, the menu cancels its registrations with the button object.

The implementation of these ideas is straightforward using the ideas and source code presented above. For brevity, we omit implementation details for this and subsequent subsystems.

4.2 Selectors

Selectors are used throughout Prism to display the contents of models (e.g., the organs belonging to the patient), and to allow elements to be added, deleted, and selected for editing. There are, however, other ways that models can be changed: Prism eases integration of new tools by making models visible, so any tool can operate directly on a model, e.g., adding, changing, and deleting organs and beams. In the face of manipulations, it is critical to keep the interface presented to the dosimetrist consistent. If an organ is added, a new item should appear in the organ menu. If the name of an organ is changed, the name displayed by the corresponding item should update. Moreover, we can meet the requirement that the user be able to change the name of an organ by allowing the user to change the button name, with a constraint that the organ name be updated accordingly. In the face of direct client accesses, all aspects of the Prism environment should be kept consistent. We now discuss how we met this requirements, focusing on selectors in particular.

4.2.1 Analysis

Figure 10 presents our analysis model of the organ selector. All other selectors are analogous. In fact, all are instances of the same class, parameterized at instantiation time to handle different kinds of objects and panels.

The primary entities in the analysis model are a set of organs, a set of easels, and a menu. To each set is associated zero or more elements. These associations change dynamically as elements are added and deleted. The two main behavioral relationships, *SM* and *ME,* maintain the one-to-one correspondences discussed above. If an organ is added to the organ set, *SM* requires a corresponding item be added to the menu. If an item is selected, *ME* requires addition of a new easel to the easel set. Conversely, if an easel is closed by the user, *ME* requires that the corresponding button be deselected; and so forth.

The relationship *ME* is dependent on relationship *SM.* This is because *ME* is responsible for creating panels for selected menu items, but a panel must be connected to the organ associated with the item selected. The association between organs and items is maintained by *SM.* Thus, when a menu item is selected, *ME* queries *SM* to find out to which organ the new panel should be attached.

The bottom half of the figure depicts the individual elements of the sets (organs, buttons, easel panels) and the behavioral relationships between them. *OB* requires associated organ and button names be equal; if either changes, the other must be updated. The arrow from the easel to the organ denotes a dependence of the easel on the organ. We basically merged the relationship between the easel display and the organ into the easel because easels are never used in the absence of organs (or generally some volume to be edited). Since the relationship is dependent, this merging made the easel dependent.

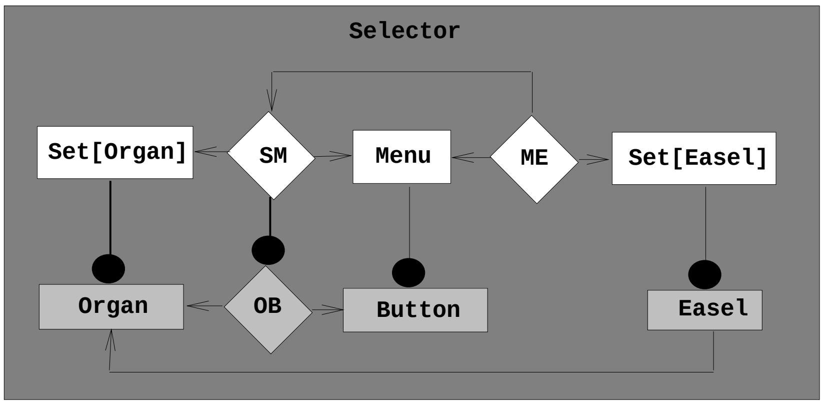

Figure 10: Simplified model of the Prism selector subsystem.

The patient model contains a set of organs. The panel subsystem presents this set and supports insertion, deletion, and selection for editing. The *SM* relationship ensures that the menu contains one button for each organ. Instances of the *OB* relationship, dispatched by *SM,* keep organ and button names consistent with each other. The *ME* relationship ensures there is one easel for each selected button. Each easel is connected to the designated organ. To find out which organ that is, *ME* queries *SM* to map a given, selected button to the organ it represents.

4.2.2 Design

To save space, we only discuss some of the key features of the selector subsystem at the design level. First, we represent instances of the *OB* relationship as mediator objects. The design for these objects is modeled on the dialbox presented earlier. The relationships *SM* and *ME* are similar to each other, so we only discuss the design of *SM*. We start the discussion with a simple but key aspect of Prism: the design of the organ set. All other sets in Prism—panels, tumors, plans, etc.—are analogous.

Sets. We represent the organ set at the design level as an ABT. The set ABT has two operations, one to insert an element, one to delete one. Each operation takes a reference to the element as a parameter. The ABT also defines events *inserted(x)* and *deleted(x),* where x is a reference to the object inserted or deleted. If a client calls the operation *insert(x),* the *inserted* event is announced if and only if x is actually added to the set, which happens if and only if x was not already in the set.

Representing multi-valued attributes as set ABTs—e.g., the organs in a patient—is one of the cornerstones of the Prism design. It provides an explicit runtime representation of dynamic entry and exit of elements into and from associations. Rather than having to maintain consistency in the face of *creation* and *deletion* of objects, we do so in the face of *insertions* into and *deletions* from sets. Doing this is easy because set ABTs announce events that can be monitored by mediators.

Relations. Designing a mediator to represent *SM* is straightforward, based on the design of the dialbox. First, the *SM* mediator references the organ set and the menu. Second, it supports four private operations, one to handle each relevant activity: insertion and deletion of organs and buttons. When the mediator is created, it receives references to the set and menu and registers these operations with their events. At runtime, it uses a "busy bit" to avoid circular updates, as in the dialbox example.

In addition, the mediator maintains a relation (in the form of a relation ABT) as part of its state. It uses the relation to record correspondences between individual organs and buttons. Each association is stored as a pair of references, one to the organ, one to the button. This relation is used both by the mediator itself, and by clients. In particular, the *ME* mediator queries this relation to find out to which organ to attach an easel when a menu item is selected.

To see how the *SM* mediator works, consider what happens when an organ is deleted. The mediator, having registered for deletion events, is invoked. It responds by checking its busy bit, which it finds clear (no update in progress). It maps the organ that was just deleted through the relation to find the corresponding button. It deletes the correspondence from the relation, then deletes the button from the menu. This invokes the mediator recursively, but the mediator finds its busy bit set, so it just returns (to the menu). Then the menu returns back to the mediator. The mediator returns to the set. Finally the insert operation completes with the set, the mediator's relation, and the menu all consistent.

Submediators. There is one additional complication. When an organ and a button are associated, we want their names to stay consistent. To implement this, the *SM* mediator uses an idiom we call *deploying submediators.* A critical advantage to the method of implementing behavioral relationships as ABT-based objects is that they can be created and can register with other objects dynamically. This is the basis for integrating tools as they are activated in Prism.

The basic idea is that in addition to maintaining a relation between two collections, a mediator also maintains a set of mediators (of a different kind) responsible for integrating related elements of these collections. In this case, *SM* creates an instance of the *OB* mediator to integrate each associated organ and button. Thus, when *SM* adds an association to its relation (see above), it creates an *OB* instance, giving it references to the organ and button. *OB* then registers dynamically with their events and keeps them consistent thereafter. When *SM* deletes the assocation, it also deletes the corresponding *OB* mediator.

The ability to integrate entities, without change, by interposing mediators eases integration as well as evolution over the life time of a system. Deploying submediators for each entry in a relation is a run-time example of the leverage gained by separating behavioral relationships, and is a key idiom used extensively in the Prism system. We discuss more sophisticated usages of this idiom in the next subsection.

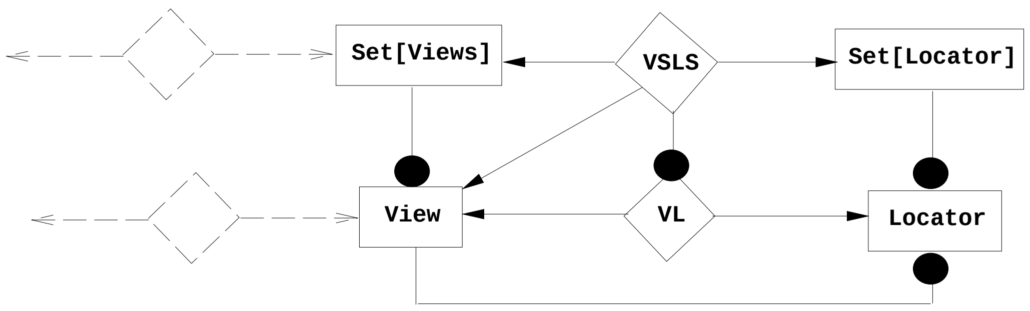

Figure 11: The Prism *locators* subsystem.

The locators subsystem contains a set of views and a set of locators. The set of views holds zero or more view instances; similarly for the locator set. The sets are integrated by a behavioral relationship *VSLS* (for view-set/locator-set). *VSLS* keeps the locator set consistent as views are added and deleted. It also inserts locators into and deletes them from the views that are to display them. Finally, it also deploys *VL* submediators to keeps locators consistent with the views they depict.

4.3 Locators

The idea of deploying submediators in correspondence with the elements in a relation provided the basis for solutions to several more difficult design problems. In this subsection we discuss how we handled locators. Recall (see Section 2.3.7) that each orthographic view displays one *locator* for every other orthographic view that intersects the given view. Thus, whenever a new view is created, a locator has to be added to eacg view intersected by the new view, and locators have to be added to the new view for each of these intersecting views.

Furthermore, the position of each locator in the view in which it is displayed has to be kept consistent with the position of viewing plane of the view it represents. If the Z position of a transverse view is changed, the locators *for* that view *in* all coronal and sagittal views have to be updated. Similarly, if the user drags one of those locators, the Z position of the corresponding transverse view must be updated. This supports a kind of crude animation: as a locator is dragged, the view it represents appears to pass through the patient model along the viewing axis. As it does this, it renders successive slices of the patient anatomy, beam locations, dose distributions, etc.

4.3.1 Analysis

Abstractly, these requirements are similar to the earlier problem: several sets have to be kept consistent as elements are added and deleted, and corresponding elements have to work together while they are related. The key idea for the locators subsystem was that we had a set of views needed to maintain a relation encoding intersections. Rather than a one-to-one correspondence, we need a behavioral relationship that maintains the *intersects* relation.

Suppose two views A and B intersect—perhaps A is transverse and B is coronal. Then the intersects relation contains the tuples (A, B) and (B, A), since each view intersects the other. Corresponding to these tuples are two locators $L_{(A,B)}$ and $L_{(B,A)}$. The first is displayed in A to represent the intersection of B with A. The second appears in B for the intersection with A. The locator system essentially consists of a set of views and a set of locators, and tuples in the latter are in one-to-one correspondence with tuples in the intersects relation. As views are added and deleted, the behavioral relationship *VSLS* maintains the intersects relation and updates the locator set accordingly.

VSLS has two additional tasks. First, when it adds a locator to or deletes it from the locator set, it also displays the locator in or removes it from the corresponding view. Thus, the mediator maintains the one-to-n relationship from views to locators, as presented in the figure. Second, each locator has to remain consistent with the view it represents. To ensure this, *VSLS* deploys submediators, one instance of *VL* to integrate each locator displayed in a view A with the view B that the locator represents.

A key benefit of the Prism architecture is that it enabled conception and development of this machinery independently of all the other relationships in which the individual views and the set of views participate. The other relationships are suggested by the dashed lines and relationship diamonds in the figure. We have already discussed some of these. The set of views is related to the menu in the views selector of the plan panel. The name of each view is related to the name displayed by the corresponding menu item. The visibility of the individual views and of the view set allow these entities to participate in and be manipulated by many different relationships at the same time.

4.3.2 Design

The design and implementation follow from the analysis model. The set of views appears in design and implementation as a visible set ABT. The views are also defined independently and visible. We changed the representation of the locator set in design to reduce the number of classes and objects. Specifically, we eliminated the set of locators in favor of each view having its own set of locators—those it displays. Locators are this distributed about the system. We did realize *VSLS* as a mediator that registers with and responds to insertion and deletion events of the view set.

When a view is added, the mediator updates the intersects relation, which it itself maintains, by comparing the new view against each view in the view set. Only views on different axes intersect, and views on different axes have different Prism class names—transverse, coronal, sagittal—so intersection is computed by comparing these names. If the names differ, the views intersect. For each intersection, the mediator creates both a locator and a submediator. It adds the locator to the set of locators in the view in which the locator is to be displayed; and it instantiates the submediator giving it references to this view and to the view the locator depicts. The submediator registers with that view and with the locator and thereafter maintains consistency between them. When a view is deleted, the mediator reverses this procedure. When locators are moved or view planes changed, the submediators maintain consistency.

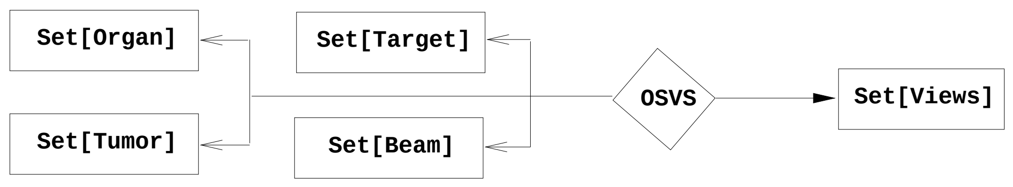

Figure 12: The Prism *graphics* subsystem.
The graphics rendering subsystem is analogous to the selector and locator subsystems. The behavior relationship *OSVS* requires that each element in each of the object sets on the left be rendered by a graphic in each of the views in the view set on the right. Although omitted in this figure, corresponding elements are related by submediators, as in the earlier subsystems.

Our behavioral entity-relationship method not only produced an analysis model that effectively separated concerns; in addition we obtained a design and implementation in which the machinery for handling the relationship between the view and locator sets is entirely separate from the machinery for the relationship between the same view set and selectors. The key properties of the analysis models—the independence and visibility of entities and the explicit representation of integration concerns—are preserved at the design and implementation levels.

4.4 Graphics

Having solved the selectors and locators problems collaboratively, Kalet applied the idea of deploying submediators to the subsystem responsible for keeping graphical renderings consistent with the treatment plans they depict. The requirement was to incrementally update all renderings—e.g., in views, easels, frames in filmstrips, etc.—when the subject changes: any set of organs, tumors, target volumes, etc. or any element of any of these sets. In Figure 3, if the easel is used to change the shape of the target volume (outlined with small squares) then at least three renderings are updated: the frame in the film strip in the easel, the transverse view in the upper right, and the beam's-eye view in the lower right. Updates are incremental in that only graphics for specific elements are changed, added, or deleted—e.g., the graphic for a single contour. With current hardware, incremental consistency maintenance is important for good runtime performance.

Figure 12 presents our analysis model. The solution is analogous to those for selectors and locators. For those subsystems we defined mediators that maintained relations between sets and that deployed submediators to integrate elements. The relation for the selector subsystem was a bijection between sets and for locators, intersections within a set. In this case the mediator maintains a *cross-product* relation: each element in each subject set (organs, tumors, etc.) must be depicted (consistently) in each view in the view set. The cross product is between the union of the subject sets and the view set.

The mediator thus creates a graphic for each object/view pair, which it inserts into the view, and it deploys a submediator between the object and the graphic to maintain consistency in the face of changes. In the case of contoured volumes—organs, target volumes, etc.—there is a submediator for each one. When the user changes the target volume contour in the easel (Figure 3) and presses *Accept* multiple submediators are invoked to update the corresponding graphics in the various renderings. If, on the other hand, a view is added to the view set, the main mediator creates m graphics and deploys m submediators—one for each object to be displayed in the new view.

The design and implementation of this model is straightforward based on the earlier examples. This structure paid off when Kalet discovered that the initial design for graphics rendering was unworkable. The problem was in using X windows [Scheifler and Gettys 86] to render multi-layered pictures, with contoured objects over background images. The mediator-based architecture made it easy to fix the problem because all graphics code was localized in the submediators: each was responsible for rendering its object in its view. We replaced the graphics pipeline without touching the *OSVS* mediator, or the object sets, or the objects, or the view sets, or the views. Moreover, these changes were independent of relationships between the view set and selector panel, between the object sets and the selectors, between the locators and views, etc. The mediator architecture effectively separated these concerns.

4.5 Beam Panel

Finally, we examine the problem of adjusting the user interface that the the beam panel presents to accommodate changes in the kind of beam being handled. Consider the beam panel in Figure 2. The key aspects of the panel are the button on the left that displays the kind of treatment machine specified as generating the beams (Clinac 4), and the part of the panel used to adjust the collimator that shapes the beam (the bottom two sliders on the panel).

If the user presses the machine button, a menu of machine types pops up. If one is selected, the machine type attribute of the underlying beam is updated. The problem is that different kinds of machines have different *collimation systems*, and different collimator types have different numbers of parameters that can be adjusted, so changing the machine type may require a different "sub-panel" for adjusting collimator settings. The sub-panel for the Clinac-4 requires two sliders. Another kind of machine needs four sliders. Another requires an entirely different interface to shape the beam aperture—one that uses a contour editor, as in the easel.

4.5.1 To Use Inheritance?

One question we asked 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 an abstract beam class with subclasses specialized by kind of collimator 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.

We had two problems with this approach. First, every ordering seemed artificial. This is because the specialization dimensions are independent. Beam types occupy a matrix, not a hierarchy. Particle type and collimation system are independent ; neither is subordinate to the other. Second, changing a beam's machine attribute could have required dynamically changing the class of the beam being edited and also dynamic replacement of beam panels. New subclasses would have to be created when machine types changed. Although CLOS does support dynamic type conversion, we found that mechanism to be too complex in contrast with the simplicity of the ideas.

4.5.2 Analysis

Instead, we defined a single beam class with attributes—slots containing objects and values—to indicate specializations. These include particle type and collimation system. We then used inheritance to model different kinds of collimators. To change the collimator type of a beam, we assign an instance of a different collimator subtype to its collimator attribute. We modeled the beam panel in the same way, with a collimator sub-panel as an attribute.

This made it easy to maintain the correspondence between machine type and collimator type 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 a 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 both the collimator sub-panel and the "submediator" that is responsible for connecting the sub-panel to the collimator object. This is just a simple adaptation of concepts presented for the subsystems discussed above.

4.6 Wrap-Up

We conclude this section with the observation that the Prism architecture and our approach to crafting it differed from common methods in two ways. First, we have developed a new way of thinking about systems—one that emphasizes both independence and visibility of entities, and the separate, explicit representation of the behavioral relationships needed to integrate them. Second, we use event mechanisms in a stylized way to map behavioral ER analysis models into structurally similar designs and implementations. We applied this approach throughout Prism, at all levels of "granularity," and in solving reasonably difficult design problems of several different kinds.

5 Development Effort

Prism is implemented in about 18,000 lines of Common LISP (CL) [Steele 90] and CLOS [Bobrow et al. 88] and 4,500 lines of Pascal. It also has 11,000 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 4,200 lines handle user interface widgets, sets, relations, and events. Prism classes take 2,700; file handling, 1,000; panels, 5,200; views and other graphical renderings, 3,100; mediators, 1,300 (except for mediators that link panels to objects, which are counted with panels); and "other" code takes about 500 lines. The code density is about 30 characters per line, with blank lines and concise documentation text included. In comparison, Kalet's first system has 47,000 Pascal lines. Kalet's second system, still in use, has 41,000 lines of Pascal, 5,000 for dose distributions. Comparing with another system, the basic functions taking 18,000 lines in Prism take about 60,000 of C [Kernighan and Ritchie] and C++ [Stroustrup 86] 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[3]

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.

We built Prism on a modest budget in person-hours and with a small project team. The effort lasted from January, 1990 to present. Ten people were involved at different times. The total effort was about 4.5 person years. Of this, requirements specification, which was done before collaboration began, took 24 person months. Design and implementation, the focus of the collaboration, took 20. Developing electron beam dose calculation code took 11.

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.

6 Discussion

A software approach that is successful in the lab can fail in practice. Approaches should be tested in realistic trials before large-scale efforts are staked on them. Prism provided a realistic trial for behavioral ER modeling and design. From Kalet's perspective, it worked. He realized a sophisticated system on a modest budget, with a few people, and ended up with small, flexible implementation.

Can we attribute success to behavioral ER modeling and design? It might, instead, be due to the use of Common Lisp, or to experience building earlier systems, or to more attention paid than in previous efforts to requirements specification, analysis modeling, and design. Some of these factors surely helped. Our collaboration was not a controlled experiment, so we can't quantify the contribution of our approach. Even without a rigorous experimental design, however, there are useful ways to evaluate the outcome.

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

First, we can ask the clients, Kalet *et al.*, whether the approach helped. They are both domain experts and have significant experience using modern (e.g., object-oriented) techniques to built radiation treatment planning systems. Nor do they have a vested interest in behavioral ER modeling and design. Their basic claim is that without the conceptual and structural benefits of these methods, meeting the functional requirements for Prism would have taken far more resources than were spent or available; and the resulting architecture would still have been inflexible, unfriendly to integration of tools yet to be developed.

Second, we can ask whether our methods helped make the clients more effective software builders. The answer is yes, but the transition was harder than expected. The problem was that the approach represents a significant change in how we model, design, and implement systems. Just as the transition from a structured to an object-oriented style requires time and a visceral understanding, so does shifting from object-orientation to behavioral ER modeling and design. Superficially, one can believe this is a small shift. Our experience with several members of this project shows that the shift is large. The approach cannot be applied mechanically, but has to be internalized.

Finally, we can evaluate the artifacts using well-accepted (although still imperfect) design quality criteria: coupling and cohesion [Stevens, Myers, and Constantine 74]; information hiding [Parnas 72]; structural continuity [Dechampeaux 93]; etc. Explicitly applying these concepts is outside the scope of this paper. The system is evidently factored into useful and interesting stand-alone entities integrated by mediators that represent intuitively clear behavioral relationships. New mediator types and instances can be deployed to integrate independent tools and objects. Our method did reconcile tight integration with a significant degree of architectural flexibility in a significant environment that is being released for hospital use.

Future Work. Prism is scheduled for hospital use in December, 1993. Its release will provide an opportunity for studying the impact of our architecture on ease of handling changes, both expected and unexpected. Using Prism as a framework for evaluating experimental tools will involve unexpected change. Observing how the architecture responds as AI-based planning engines and other prototype tools are integrated will provide a good test of our approach in the face of unexpected change.

Acknowledgments. Jonathan Unger kindly provided a great deal of technical assistance.

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.

[Dechampeaux 93] D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development,* (Reading, Massachusetts: Addison Wesley), 1993.

[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 and others, "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.

[Griswold 91] W.G. Griswold *Program Restructuring to Aid Software Maintenance,* Ph.D. Dissertation, Technical Report 91–08–04, Department of Computer Science and Engineering, University of Washington, August, 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.

[Johnson 92] R.E. Johnson and V.F. Russo, "Reusing Object-Oriented Designs," University of Illinois at Urbana-Champaign, Technical Report UIUCDCS-R-91-1696, 1991.

[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.*

[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.

[McCabe 91] T. McCabe. Programming with Mediators: Developing a Graphical Mesh Environment. Masters Thesis, University of Washington. 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 will 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.

[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.

[Stevens, Myers, and Constantine 74] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal 13,*2, pp. 115–39, May, 1974.

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language,* (Addison-Wesley: Reading, Massachusetts), 1986.

[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, Forthcoming in 1994.

[Taylor 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.