Mediators:
Easing the Design and Evolution of Integrated Systems

Kevin J. Sullivan

Technical Report 94-08-01

Department of Computer Science and Engineering
University of Washington

Mediators: Easing the Design and Evolution of Integrated Systems

by

Kevin J. Sullivan

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1994

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree______________________________________

Date____

University of Washington

Abstract

Mediators: Easing the Design and Evolution of Integrated Systems

by Kevin J. Sullivan

Chairperson of the Supervisory Committee: Professor David Notkin
Department of Computer Science
and Engineering

People benefit from tightly integrated systems that can be designed, realized and evolved affordably. Unfortunately, common software design methods do not easily accommodate requirements for tightly integrated systems. Indeed, when used to meet such requirements, common methods tend to yield unnecessarily complex structures that complicate design and realization and that inhibit subsequent evolution. After substantiating this claim, I present the *mediator method* as a solution. This method combines *behavioral entity-relationship (ER) modeling* for design with a *mediator* approach to implementation. The mediator method is better than common methods for designing, realizing, and evolving many integrated systems. I support this claim both by rational argument from simplifying models and by carefully reflecting on software development experiences in which the method was used. One of these efforts led to the design of Prism, a system used to plan radiation treatments for cancer patients, now in clinical use at the University of Washington Cancer Center. I present Prism as a case study on the use of the mediator design method. The mediator method represents a contribution of significant value because it allows ordinary software developers to design, realize, and evolve more effective integrated systems more affordably—with common programming languages and without costly or restrictive new mechanisms.

Table of Contents

List of Figures

Acknowledgements

I have many people to thank and acknowledge. My advisor David Notkin was generous beyond any reasonable expectation with time, advice, ideas, support, and patience. I thank Ira Kalet and the Department of Radiation Oncology for allowing me to design Prism, the department's new radiation treatment planning integrated environment, as a way to test my research results. The other members of my dissertation reading committee, Craig Chambers and Alan Borning, provided valuable feedback, guidance, and support. I thank Steve Hanks and Eve Riskin for being on my supervisory committee.

Among current and former graduate student colleagues, Bill Griswold, Tom McCabe, John Maloney, Kingsum Chow, Mike Van Hilst, and Gail Murphy were especially helpful with ideas and editing. Responsibility for remaining errors and omissions rests entirely with me. The Department of Computer Science and Engineering has been wonderful: faculty, staff, students, and visitors. I also wish to acknowledge support I received under a fellowship provided by GTE.

Finally, I thank my family and friends. My parents Bill and Marilyn Sullivan have encouraged me for many years. Molly Knox and Parker MacCready were especially good friends while I was in graduate school. A very warm thanks go to my companion Anne Knox, who supported me with kindness and generousity through my final year of dissertation writing.

Chapter 1

Introduction

Software designers often have to integrate the behaviors of separate software components. The problem I address in this dissertation is that common software design methods unnecessarily complicate the design, realization, and evolution of such integrated software systems. With common methods, integration generally requires changes to the components to be integrated or to their clients—changes yielding unnecessarily complex software structures. As evolving integration requirements continue to be met, unnecessarily structural complexity accumulates, driving up the cost of change. Eventually—often quickly—the cost becomes prohibitive of further integration or evolution [Lehman 80].

This dissertation makes two contributions to the software engineering field. The first is a new design method—the *mediator method*—that overcomes the problems with common methods without costly new mechanisms. The mediator method views both behaviors and the behavioral relationships needed to integrate them as first-class abstractions; and it implements both as first-class, imperatively programmed components—instances of what I call *abstract behavioral types.* The second contribution is an analysis of software design methods that helps to explain how these methods affect the ease of designing, realizing and evolving integrated systems. The analysis characterizes the design spaces in which the methods operate, key design choices that they promote, and certain static and dynamic structural properties that result from the use of these methods.

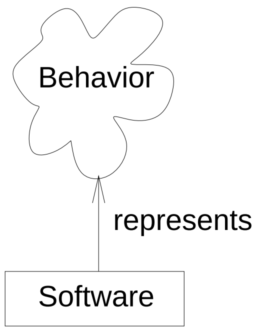


Figure 1.1: Concrete software artifacts represent abstract behaviors.

1.1 Integration

Software represents behavior, much as numerals represent numbers—as diagramed in Figure 1.1. The client of a software system ultimately benefits from its abstract behavior. How that behavior is realized by the software representation is not directly relevant to the client. Yet the software engineering properties of the representation affect the value obtained by the client by affecting the cost of the software. This dissertation addresses the impact of integration on these properties, and hence on cost and, ultimately, value.

By *integration* I mean a combination of separate parts into a whole in which the parts remain distinguishable. Both behaviors and software artifacts are subject to integration. Behavioral integration combines simpler behaviors into a more complex behavior in which the simpler ones remain distinguishable. Software integration similarly combines simpler software representations into a more complex representation.

To make the idea of behavioral integration concrete, consider the integration of two behaviors familiar to most programmers. An editor behavior supports the creation and editing of source code. A compiler behavior supports the translation of source code into object code. Instead of two separate behaviors, a programmer may want one behavior in which the two are integrated: an integrated programming environment with the property that whenever the editor saves a source code file, the compiler runs to update the corresponding object code. The behaviors are distinguishable but they work together. (See the top part of Figure 1.2.)

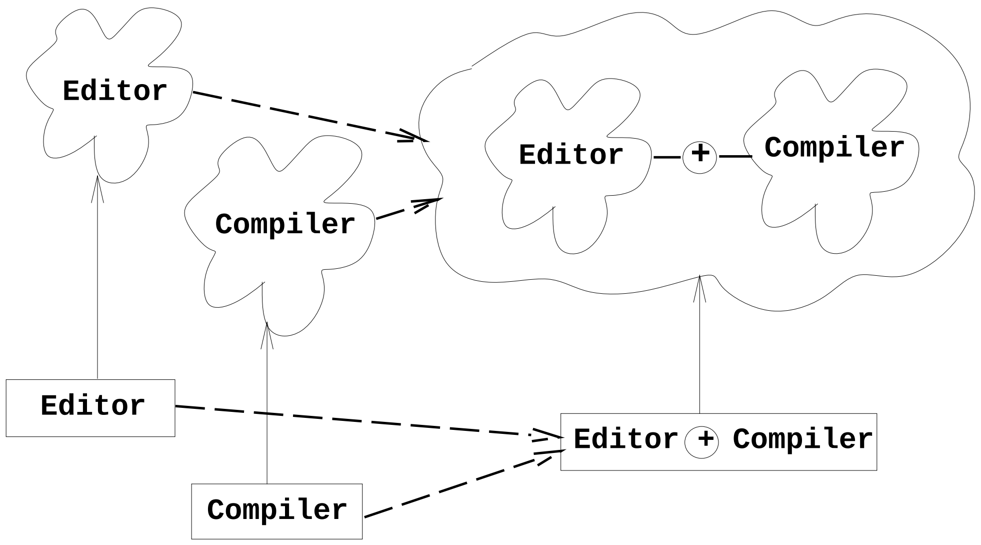


Figure 1.2: Behavior and software integration.
The top part of this figure depicts the integration of two behaviors, denoted by the clouds. The behavioral relationship $\oplus$ indicates that when the editor saves a source file the compiler updates the corresponding object code. The bottom part of the figure depicts the integration of software artifacts representing the behaviors, denoted by rectangles. The result is an integrated software artifact representing the integrated behavior.

To make the idea of software integration concrete, consider how one would implement this integrated behavior if one already had representations of the separate behaviors. If, for example, one has source codes for the editor and compiler, then integrating these software artifacts is an obvious possibility. One could produce source code for the integrated behavior by integrating the sources for the separate behaviors. Indeed, this approach is often the only economical way to represent integrated behaviors. (See Figure 1.2.)

Although behavioral integration is independent of software integration in theory (since behaviors are independent of particular representations), behavioral integration and software integration are tightly intertwined in practice. In particular, software designers often represent integrated behaviors as integrated software systems in which components representing the constituent behaviors are integrated to produce a representation of the overall, integrated behavior. This dissertation addresses the connection between behavioral and software integration.

1.2 Software Design Methods

One of the primary goals of this work is to characterize how the software engineer's choice of design method affects the difficulty of designing, realizing, and evolving integrated behaviors and integrated software artifacts to represent them. By a *design method* I mean an approach to decomposing behaviors into or composing them from simpler behaviors; a way of representing behaviors as software artifacts; and a way of integrating software artifacts in order to integrate the behaviors that they represent.

1.2.1 Basic Building Blocks

All of the design methods that I consider in this work take *objects,* broadly speaking, as basic behavioral and representational building blocks. By an *object* I mean either an instance of an abstract type or a software representation of such an instance.

The design methods differ from each other in two important dimensions: the kind of abstract type used to design and represent behaviors; and the way in which given representations are integrated to integrate their respective behaviors.

One kind of abstract type that is widely used to characterize and represent behaviors is the abstract *data* type (ADT). An ADT defines a class of objects in terms of the state spaces of the objects, possible initial states of objects, and operations that can be applied to observe or change the state of objects [Liskov and Zilles 75, Meyer 88].

I will argue that there is a second kind of abstract type: one that augments the ADT with the means for one object to implicitly extend certain behaviors of others. A mechanism that supports implicit extension is *implicit invocation*, or event notification [Garlan and Notkin 91]. Using such a mechanism, an object m extends the behavior of an object n by registering an operation to be invoked by an *event* announced by n. When n announces the event, the operation is invoked implicitly. In addition to operations, objects of this kind are characterized by events that make some behaviors "audible" as event announcements that other objects can listen for and to which they can respond.

1.2.2 Putting Blocks Together

Another way design methods differ from each other is in the way they integrate given software artifacts. In one approach, based on hierarchical composition of ADTs [Liskov and Zilles 75], one represents an integrated behavior as an object that encapsulates and manages objects representing the behaviors to be integrated. A second approach integrates "peer" objects by having them communicate through implicit or explicit invocation. One represents an integrated behavior as a network of objects that represent the constituent behaviors and that communicate with each other as needed.

1.3 Common Design Methods

The first method I consider uses hierarchical composition to achieve integration. The other three methods structure systems as networks of communicating objects. This section describes these common design methods and discusses how they complicate the design, realization, and evolution of integrated systems.

1.3.1 Encapsulation

The encapsulation method takes the ADT as its basic building block and takes a hierarchical encapsulation approach to integration. It represents integrated behaviors as instances of what I will call "encapsulating wrapper" types. The behaviors of instances of given types are integrated by incorporating them as hidden components of a new "wrapper" type. To make the behaviors of these now hidden instances accessible, the wrapper *promotes* the operations of the encapsulated objects: it exports operations mimicking those of the encapsulated objects.[1] The promoted operations mimic the encapsulated operations by invoking them. The promoted operations may also extend those operations, to ensure proper integration, by invoking the operations of other encapsulated objects.

[1] I use the term *promotion* in the same way as it is used to describe similar structures in specifications written in Z [Spivey 89].

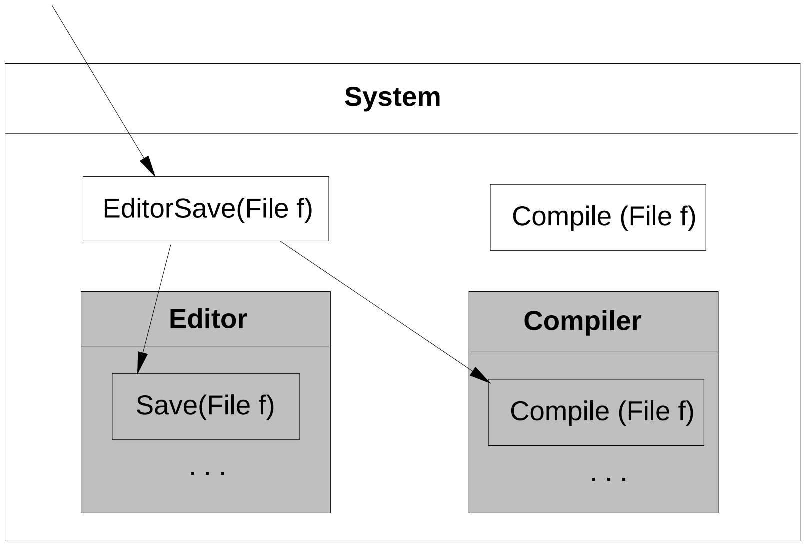


Figure 1.3: The encapsulation approach to behavioral integration.
The wrapper object *System* encapsulates editor and compiler objects, shaded to indicate they cannot be accessed directly by clients. Arrows indicate invocations. Encapsulation forces clients to use the wrapper, ensuring integration.

To make this clear, suppose one has ADT representations of compiler and editor behaviors, and one wants a system in which the editor and compiler work together—as described earlier. The encapsulation method is often used in such situations. Figure 1.3 illustrates. One defines a new type that aggregates and encapsulates editor and debugger type instances. Aggregation incorporates the given behaviors as parts of a larger one. Encapsulation prevents direct access to the editor or compiler. This is needed because direct access could leave the behaviors not properly integrated: the editor could be made to save a source file without the compiler being run. To compensate for hiding the editor and compiler, the wrapper promotes their encapsulated operations. Thus, if the editor has an operation *Editor.Save,* the wrapper, *System,* exports *System.EditorSave.* The implementation of this operation calls *Editor.Save* (making the editor behavior accessible as a constituent part of the integrated behavior); then, before returning, it calls *Compiler.Compile* to regenerate the corresponding object code, satisfying the integration requirement.

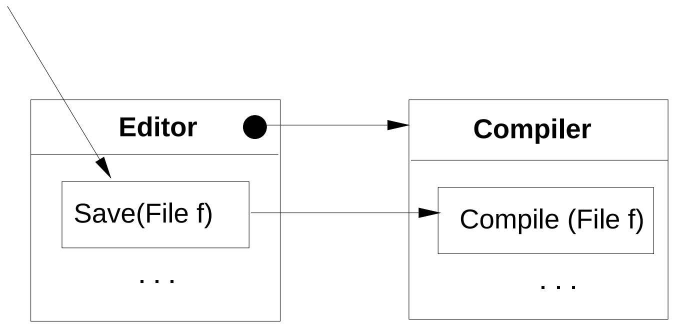


Figure 1.4: The hardwiring approach to behavioral integration.
The dot in the editor object and the arrow from it depict a reference from the editor to the compiler object, which is used by the editor to call the compiler.

This approach has the advantage of not requiring changes to the given artifacts. Instances of the editor and compiler types are used without change. Unfortunately, their clients have to change. One can no longer use the editor save operation directly, but must use the wrapper instead. This method also has the disadvantage of producing monolithic types. The wrapper ends up representing both the editor and compiler behaviors and the behavioral relationship that integrate them. When more behaviors and relationships are required, this approach leads to large wrappers whose complex interface and implementation structures complicate integration and evolution. The encapsulation design method thus scales poorly as integration requirements evolve.

1.3.2 Hardwiring

The hardwiring method uses explicit invocation as a communication mechanism to integrate the behaviors of ADT-based objects. This is a very common method. To integrate objects, one changes the objects to make them call each other.[2] For example, if given editor and compiler objects and an integration requirement as above, many programmers would change the editor save operation by inserting a call to the compiler's compile operation. Figure 1.4 illustrates.

[2]Object-oriented languages allow these changes to be isolated in subclass or subtype definitions. The inability of subtyping and subtype polymorphism to overcome problems with common design methods is discussed in Chapter 8.

A benefit of this approach is that clients of the given objects do not necessarily have to change when objects are integrated. There is no encapsulation, nor do the object interfaces necessarily change. The key changes are in the implementations of existing operations, from which clients are generally insulated by information hiding mechanisms.

Yet, this method suffers from serious problems. The first is that the objects whose behaviors are to be integrated themselves have to change. Second, these changes complicate the objects, and specifically create reference dependencies between them. The objects must invoke each other and therefore must reference each other (or at least intermediate objects). Moreover, as integration requirements evolve, each object may have to manage interactions with more and more other objects. Third, this method encourages neither separate conception nor representation of behavioral relationships. Instead, the code and data needed to implement required relationships are distributed among the objects to be integrated. This makes it hard not only to find the representations of relationships but to update them when behavioral relationships change. The code for one relationship may be commingled in the code of several objects, along with code for other relationships in which the same object participates. The complexity of objects grows in proportion to the richness of the integration of their behaviors. Changes to existing relationships or the addition of new ones requires handling of ever more complex objects. The cost of change mounts quickly as integration requirements evolve.

1.3.3 Implicit Invocation

The implicit invocation design method is similar to hardwiring, but provides the designer with a different way to structure dependencies between communicating objects. In addition to explicitly invoking each other, objects can register to be invoked by each other's events. Instead of the editor calling the compiler, the compiler can register with the editor to be notified when the editor announces a file-saved event. The compiler object depends on the editor object, even though it is the behavior of the editor that affects the compiler. See Figure 1.5.

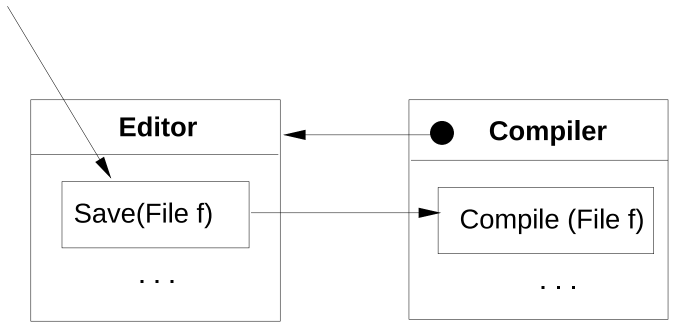


Figure 1.5: The implicit invocation approach to behavioral integration.

Implicit invocation is an old idea. Many systems support specialized event mechanisms, including Smalltalk-80 [Goldberg and Robson 83], LOOPS [Stefik, Bobrow, and Kahn 86], APPL/A [Sutton, Heimbigner, and Osterweil 90], and many others. Despite this long history, a precise characterization of implicit invocation and its benefits has only recently emerged [Garlan and Notkin 91, Sullivan and Notkin 92]. The basic advantage of implicit invocation is that it provides engineers with the flexibly to choose the orientations of reference dependences between components that invoke each other.

Despite this added flexibility, the problems with the implicit invocation method are basically the same as with hardwiring—with some dependencies reversed. Achieving integration compromises the independence of the given objects. It is no more appropriate for the compiler to reference the editor than the other way around. As integration requirements evolve, the level of unnecessary complexity mounts. Although details differ, the static and dynamic structural properties are about the same as with hardwiring.

1.3.4 Broadcast Message Servers

The broadcast message server (BMS) method, pioneered by Reiss [Reiss 90], specializes the implicit invocation method by using a distinguished broadcast message server component to coordinate communication among other components. Each component informs the BMS of the operations it supports; sends messages to the BMS to invoke operations of other components; sends messages to the BMS to announce its own events;

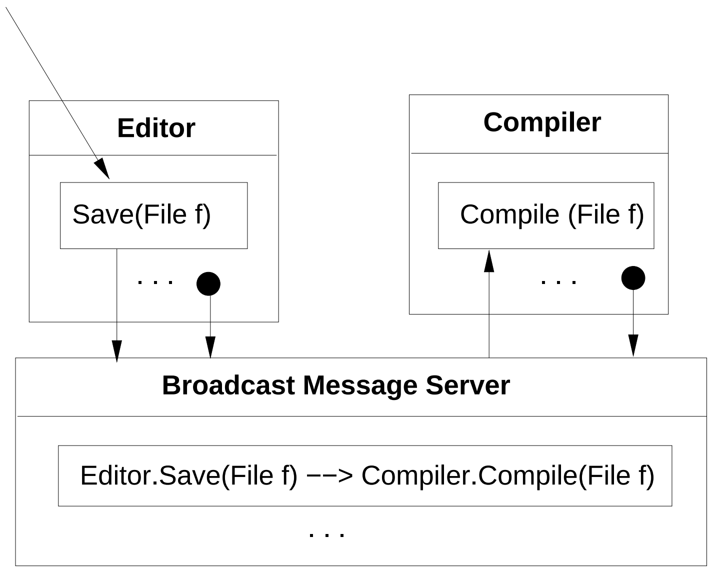


Figure 1.6: The broadcast message server approach to behavioral integration. The BMS records registrations of operations with events. The compiler registered a "trigger" as depicted, causing BMS to implicitly invoke *Compiler.Compile(f)* when the editor explicitly invokes the BMS to announce that it saved the source code file f.

and registers with the BMS to be notified of events announced by other components. A BMS supports designs similar to blackboard systems, in which multiple clients are notified of changes made to central information repositories. Figure 1.6 illustrates.

A benefit of this approach is that the added level of indirection between components decouples component interfaces from implementations, providing for a kind of polymorphism that enables "plug compatibility" of tools. One compiler or configuration manager can be replaced by another so long as the message protocols are compatible.

Unfortunately this method has problems similar to those of hardwiring and implicit invocation. Integration requirements are met by changing existing components to make them communicate with each other; and these changes compromise the simplicity and independence of the components. Although the components do not have to reference each other directly, they come to depend on each other's interfaces and protocols. Nor is there an impetus to represent behavioral relationships abstractly. This problem is addressed in Forest [Garlan and Ilias 90], but not in great generality.

There are other problems, too. The BMS is a monolithic component, representing the global union of the interfaces of all system components. This may have serious results. It may be hard to define a system with two instances of the same kind of component because the two interfaces would conflict within the shared BMS. Another problem is that the events and operations that objects support may not be declared, making it hard to understand the abstractions that objects support [Meyers 91].

1.4 The Mediator Method

This dissertation presents a new design method that largely overcomes the problems with common design methods to ease the design, realization, and evolution of integrated systems. The basic idea is to view both behaviors and behavioral relationships as first-class abstractions and to represent both using first-class objects. It is especially important to consider and to represent behavioral relationships as objects separate from the objects whose behaviors they relate, separate from clients of those objects, and separate from other objects representing other behaviors and relationships in the same system. An integrated behavior is represented as a set of independent behaviors integrated in a network of externalized behavioral relationships. Recalling Chen's entity-relationship data modeling [Chen 76], I call these networks *behavioral entity-relationship (ER) models.*[3]

1.4.1 Applying the Method

The top part of Figure 1.7 depicts a behavioral ER model of the behavior of the integrated programming environment. Having produced this model, the second step of the mediator method produces an implementation whose structure follows that of the behavioral ER model. The bottom part of Figure 1.7 illustrates this idea. Each behavior and behavioral relationship in the model is realized by a corresponding, first-class implementation object. The behavioral ER model as a whole is realized by the corresponding collection of implementation objects.

[3]I discuss the connection of behavioral ER models to ER data models in Chapter 9.

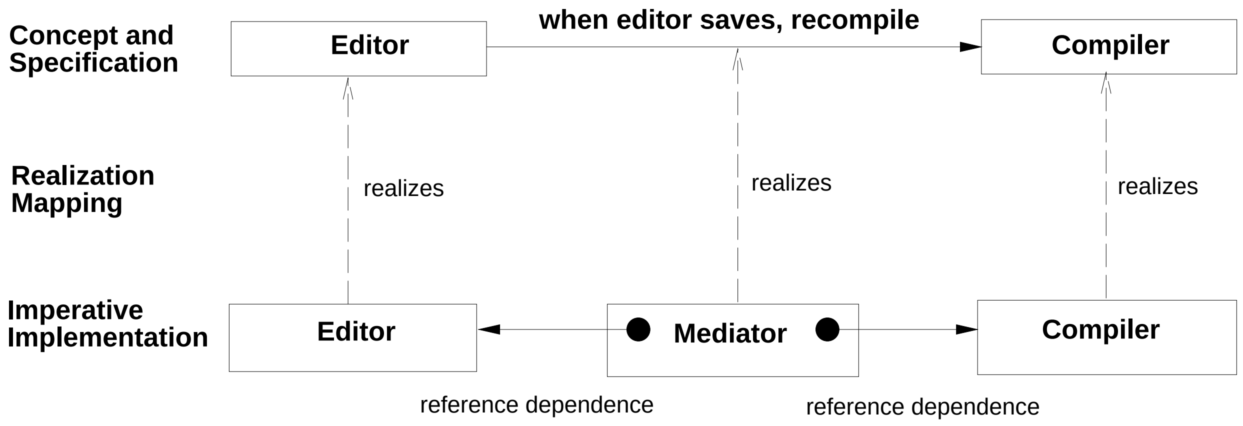


Figure 1.7: Schematic of a mediator-based realization of the programming environment. The top part of the figure presents a behavioral ER model of the behavior of the integrated environment, decomposing it into two behaviors (editor and compiler) and a behavioral relationship that defines how the behaviors work together. The bottom part of the figure depicts the structure of a mediator-based implementation of the behavioral ER model.

Not only is the static structure of such an implementation similar to that of a behavioral ER model, but also its dynamic structure: its pattern of structural change as evolution—or even execution—occurs. Just as behaviors in behavioral ER models are integrated by adding relationships, integrating the behaviors of given implementation objects is done by adding objects that represent the relationships needed for integration. These behavioral relationship objects are what I call mediators. They are distinguished by their purpose—to integrate the behaviors of otherwise independent objects—and by a stylized used of implicit and explicit invocation mechanisms (which I discuss below).

One insight of this work is that objects based on ADTs communicating by explicit invocation alone do not support this approach. One cannot obtain the desired integration and the desired independence at the same time. Nor does subtype polymorphism provide much relief, as discussed in Chapter 8.

Instead, the objects on which I base mediator implementations are instances of what I will call *abstract behavioral types* (ABTs). In contrast to an abstract data type, which characterizes a class of objects primarily in terms of the operations that can be applied,

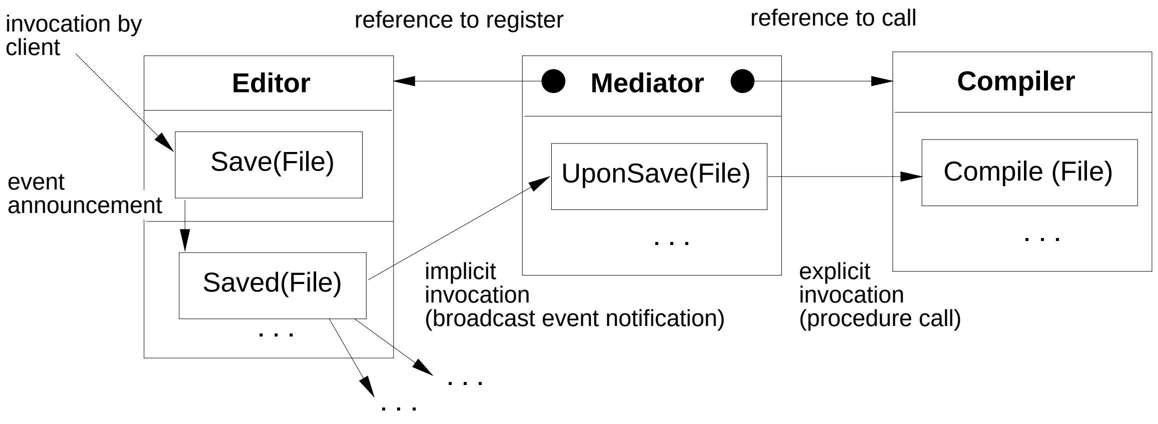


Figure 1.8: Mediator/ABT implementation of the integrated programming environment. The editor is now conceived as an object with an operation for saving files and an event, a broadcast announcement of which indicates the saving of the indicated file. The compiler is similar, but without an event in this case. The mediator responds the editor event by calling the compiler. This externalizes and modularizes the required behavioral relationship.

an abstract behavioral type characterizes a class of objects both by the applicable operations, and by the events announced by the objects to make selected, abstract behaviors "audible" so that other objects can listen for and respond to them.

Figure 1.8 illustrates how such objects are used in a mediator implementation of the behavioral ER model for the programming environment. The editor behavior is realized by an *Editor* ABT with an operation to save a file and an event *Saved* that is announced whenever the save operation succeeds. The behavioral relationship is realized by the mediator. Having registered its *UponSave* operation with this event, it is invoked whenever the event is announced. When so invoked, the operation explicitly invokes the compiler to regenerate the object code. The key is that the independent behaviors are realized by independent objects and that the behavioral relationship is realized by a separate mediator external to the object whose behaviors are integrated.

1.4.2 Properties of the Method

In general, the fact that mediators are first-class objects allows one to realize complex behavioral relationships as mediators. The heart of the mediator method lies in realizing complex behaviors and relationships as abstract, modular, imperatively programmed components. In contrast with common design methods, this method overcomes the problems of integration and evolution because it promotes software modularizations that anticipate evolving integration requirements, in analogy with Parnas's information hiding design method, which focused on anticipating the evolution of data representations and algorithms [Parnas 72].

To see better how this method eases evolution consider two evolutionary changes. The first is a change in the way that the editor and compiler work together, so that the compiler regenerates object code when the editor saves a file only if the system load is low [Garlan and Ilias 90]. To handle this change in the specification requires changing only the behavioral relationship. We do not have to change the editor or compiler or their clients. The corresponding implementation change is a localized update of the mediator object.

Second, suppose that we want to integrate a new tool, a configuration manager, so that when the editor saves a source code file a copy of the source code is checked in to a repository. We model this change by adding a configuration manager behavior and another behavioral relationship to integrate this tool with the editor. In the implementation, we similarly add an independent configuration manager object and a mediator to integrate it with the editor.

In summary, behavioral ER models provide a framework that helps the software engineer to think clearly about behavioral integration and to express these thoughts clearly in efficient, imperative computer programs. Because behavioral ER models orchestrate a graceful evolution of integration requirements, and because mediator implementations preserve the static and dynamic structures of behavioral ER models, such implementations are themselves robust with respect to integration and evolution. This method does not depend on special-purpose, costly, unfamiliar mechanisms, nor on specific languages. Whereas the negative contribution of this work shows that common design methods inherently complicate design, integration, and evolution, the positive contribution—the mediator method—solves these problem for a wide range of systems.

1.5 Evaluating Design Methods

Validating the claims made in this work requires a careful evaluation of design methods. I take a two-pronged approach. First, I argue intellectually that common methods complicate design, integration and evolution and that the mediator method should perform better. Second, to substantiate the claim that the mediator method is better, I report on experiences using it.

Chapters 2 through 5 present the intellectual case as a rigorous evaluation of design methods with respect to ease of integration and evolution. Chapter 2 defines an *evolutionary scenario* that characterizes behavioral integration and evolution. Chapter 3 presents a framework for modeling design methods and the structures they produce. Chapter 4 uses the framework to explain why common methods handle evolving requirements poorly. Chapter 5 uses the framework and scenario to present the mediator method and to explain how it overcomes this problem.

The question remains as to whether the mediator method solves these problems in the real world—whether problems were overlooked. Chapters 6 and 7 address this concern by discussing systems built using mediators. After briefly summarizing the intellectual arguments, Chapter 6 discusses prototype and production systems in which mediators were systematically employed. Chapter 7 expands on one, the Prism radiation treatment planning system, presenting it as a case study [Sullivan, Kalet and Notkin 93].

Finally, Chapter 8 evaluates this work: whether the problem is important, whether the data support the conclusions, the validity and limitations of the methods used, and so forth. Chapter 9 discusses connections to related work. Chapter 10 summarizes the dissertation and outlines directions for future work.

Chapter 2

Integration and Evolution

Discovering how software design methods affect the ease of designing, realizing and evolving integrated systems requires a careful characterization of integrated behaviors; the ways in which integrated behaviors evolve; the design methods to be studied; properties of software structure that are affected by design methods and that are relevant to ease of design, integration, and evolution; and the relation of these structures to the difficulty of these tasks. This chapter addresses the first two of these concerns, integrated behaviors and their evolution.

I characterize behavioral integration and evolution using an *evolutionary scenario* as a prototypical example. The scenario comprises three simple, initially independent behaviors subjected to ongoing changes in integration requirements. Behaviors are integrated; the relationships that define how the behaviors work together are changed; and behaviors and relationships are removed.

By abstracting from application-specific complexity while preserving the essential aspects of integration and evolution, the scenario characterizes key aspects of evolving integration requirements in the real world. The scenario thus provides a benchmark for evaluating how well design methods handle evolving integration requirements. I use the scenario this way in Chapters 4 and 5. The scenario also circumscribes the scope of this work by characterizing the kinds of behaviors, integration and evolution that I consider.

2.1 A Motivating Example

The evolutionary scenario begins with three simple switch behaviors, each used by a corresponding client. The scenario then imposes a sequence of integration requirements that call for the switches to work together, in different ways at different times, as they are used by their clients.

Even though this scenario involves only the coordination of binary digits (switches), it still captures key aspects of integration and evolution. To give a sense for how the scenario relates to realistic systems, I present an example adapted from the literature [Rumbaugh et al. 91, pp. 99–100]. The switch-based scenario directly models this example, but without the trappings of the specific application domain.

Imagine a shop in which automotive subsystems are designed and integrated. One system is a transmission. It can be engaged or disengaged. The second is a door locking device. When activated it locks all four car doors. The third system is an ignition interlock. When on (and when integrated with the ignition) it prevents the ignition from starting; when the interlock is off, the ignition is enabled to start the car. The ignition is yet another system that can be turned on and off. These are the separate behaviors.

The transmission and door-locking system are the first to be integrated. When the transmission becomes engaged, the door-locking system is to be activated to lock the doors. The doors can be freely locked or unlocked thereafter without affecting the transmission. Next, the transmission is integrated with the ignition interlock: the interlock is to be off (so the ignition can start the car) if and only if the transmission is disengaged. Subsequently, we are asked to add an interlock override, as a special kind of safety feature. If the override is turned on, the ignition is allowed to start the car even with the transmission engaged. This enables moving the car for short distances using the battery to power the starter motor. Unfortunately, to cut unexpectedly high costs, management decides to cut the automatic door locking feature; so, that system has to be removed from the car. These are the evolving integration requirements.

2.2 An Evolutionary Scenario

To reason generally about integration and evolution on the basis of an evolutionary scenario demands a precisely defined scenario (its structure, behaviors, and integration requirements); that the scenario abstract from irrelevant details; and a clear connection of the special cases in the scenario to the general case in practice. The rest of this chapter handles these tasks.

An evolutionary scenario defines a partially ordered family (a directed, acyclic graph) of behaviors for which software representations are to be built. See Figure 2.1. The nodes represent requirements for behaviors to be realized; the edges represent steps in the evolution of these requirements. A particular edge indicates that the behavior at the source node is modified, giving the behavior at the destination node; or that the behavior at the source is incorporated as a constituent into the integrated behavior at the destination. In the scenario presented below, a node on which one edge is incident represents a behavior differing from the one at the origin in one of two ways: either by a change in the way that constituent behaviors are integrated, or by the removal of a constituent behavior. Incidence of two edges on a node indicates that the behaviors at the source nodes are integrated with each other in the behavior at the destination.

The scenario presented below starts by defining three separate behaviors. The switches $B1$, $B2$, and $B3$ are toggled by respective clients. (See Figure 2.1.) The next step in the evolutionary sequence integrates $B1$ and $B2$ in a behavioral relationship $\oplus$ that defines how the switches work together when either one is toggled. The details are presented below. Next, $B3$ is integrated with $B1$ (which is now part of the system $B1 \oplus B2$) in the behavioral relationship $\otimes$. Next, the relationships $\otimes$, which defines how $B1$ and $B3$ work together, is slightly changed, yielding a new relationship $\otimes'$. Finally, $B2$ is removed from the system, along with the relationship $\oplus$ integrating $B2$ with $B1$. Note that this scenario models the static and evolutionary structure of the automotive example, above—except for the ignition, which is addressed at the end of this section.

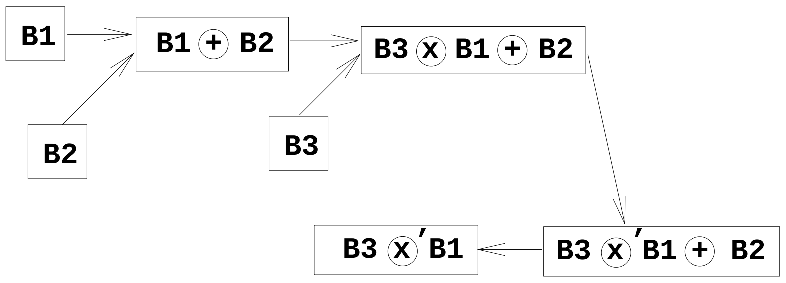


Figure 2.1: Graphical representation of a simple evolutionary scenario.
Each labeled box represents a required behavior. The directed graph represents the steps in the evolution of the requirements. Initially, three behaviors $B1, B2$ and $B3$ are required. Then $B1$ and $B2$ are integrated in the behavioral relationship $\oplus$. Next $B3$ is integrated in the behavioral relationship $\otimes$ with $B1$ (now a constituent of $B1 \oplus B2$.) The next step changes the way that $B1$ and $B3$ work together by changing $\otimes$ to $\otimes'$. Finally, the behavior $B2$ and the relationship $\oplus$ are removed from the system. Clients are not depicted in this figure.

2.2.1 An Initial Family of Behaviors

The three central behaviors in the scenario are switches, $B1$, $B2$, and $B3$. To be precise about their behaviors, I model each as an instance of an abstract type called *Switch.* The state space of an instance has two values, *On* and *Off*. The initial state is *Off*. Three operations can be applied to instances. *IsOn* is always applicable. It returns *true* for an instance that is in the *On* state, and *false* for one that is *Off*. *TurnOn* is applicable only if the switch is *Off*. It turns the switch *On*. *TurnOff* is symmetrical.

Each switch has a client, respectively $C1$, $C2$, and $C3$. Each client occasionally checks the state of its switch using *IsOn*, then toggles the switch using *TurnOn* or *TurnOff* as appropriate. I assume that clients apply these operation pairs in a way that is atomic.[1] I introduce clients because in practice the impact of integration on clients can be large. To characterize how design methods accommodate the evolution of integration requirements, it is necessary to characterize how clients are affected by integration. An editor user may be affected when the editor is integrated with a compiler. The clients in this scenario abstractly represent such client concerns.

[1] One may even view this situation as a single client using distinct switches: $C1 = C2 = C3$.

2.2.2 Adding an Asymmetric Behavioral Relationship

The first integration step in the scenario calls for the integration of $B1$ and $B2$ in the behavioral relationship $\oplus$. I define this relationship $\oplus$ as requiring $B2$ to be turned on whenever $B1$ is turned on. This relationship imposes an "edge triggering" behavior on the system. Whenever $B1$ makes a transition from *Off* to *On* (owing to the invocation of $B1.\mathit{TurnOn}$), $B2$ must also be *On* before $B1.\mathit{TurnOn}$ completes. $B2$ can then toggle independently of $B1$, and $B1$ can be turned off without affecting $B2$; but as soon as $B1$ is turned on again, so must $B2$.

This relationship characterizes the class of behavioral relationships that require asymmetric propagation of behavioral effects between subsystems. In practice, many integration requirements call for such asymmetric propagation. In the automobile, engaging the transmission requires the doors to be locked—but not vice versa. In the programming environment, saving the source code causes object code to be updated. In the Smalltalk-80 MVC [Krasner and Pope 88], changing a model updates its views.

2.2.3 Adding a Symmetric Behavioral Relationship

Another common class of integration requirements call for behavioral relationships involving symmetric or multi-directional propagation of effects—often to maintain an invariant over several objects. In the automotive case, the ignition interlock can be off if and only if the transmission is disengaged; so if either is toggled, the other must be toggled, too. In a graphical user interface text window, the position and size of the text and the position and size of the scroll bar thumb are kept consistent as either is changed.

This class of behavioral relationships is abstracted in the second step of the evolutionary scenario, where $B3$ is integrated with $B1$ in the new behavioral relationship $\otimes$. This relationship requires $B1$ and $B3$ to operate in tandem: whenever either one is turned on or off, the other must be turned on or off before the initiating operation completes.

Note that $B3$ is being integrated here with $B1$ which is already integrated with $B2$. In addition to characterizing relationships that require symmetric or multi-directional propagation of effects, this example also characterizes systems in which behaviors participate in multiple behavioral relationships. $B1$ works with $B2$ and with $B3$ in different relationships. To illustrate how this system operates, suppose $C3$ turns on $B3$. The relationship $\otimes$ requires that $B1$, which was off, be turned on. That in turn requires $B2$ to be turned on if it is not already on—owing to the presence of $\oplus$. Analogously, in the automobile, if the transmission is engaged the doors are locked and the interlock is set.

Finally, the scenario as defined so far characterizes integration in the *evolutionary* dimension. The set of relationships in which the given switch behaviors participates has changed twice so far, and the set of relationships in which the behavior $B1$ participates has also changed twice. Because changes in the sets of relationships in which behaviors are integrated are fundamental, software design methods should accommodate this kind of change without undue effort.

2.2.4 Changing a Behavioral Relationship

Not only do sets of behavioral relationships tend to change over time, but individual behavioral relationships change, too. The next step of the scenario abstracts evolution of this kind. To illustrate, consider changing the behavioral relationship $\otimes$ to $\otimes'$, a new relationship with two states, eager and lazy. When in eager mode, $\otimes'$ behaves like $\otimes$: $B1$ and $B3$ toggle in tandem. When in lazy mode, however, $B1$ and $B3$ are permitted to toggle independently, as if they were not integrated. When the relationship $\otimes'$ itself is toggled from lazy to eager state, it reestablishes consistency of $B1$ and $B3$ if necessary by turning $B3$ on or off. Thereafter, it makes the two switches toggle in tandem.

This part of the scenario illustrates one characteristic way in which behavioral relationships change: in the tightness with which consistency constraints are maintained. The addition of a safety override in the automotive system is similar: when the override is engaged, the linkage between the transmission and the ignition interlock is relaxed. Another way in which a behavioral relationship might change is in the way in which an underspecified constraint is resolved—e.g., how changes to "views" are reflected as changes are made to models from which views are derived. Changes in behavioral relationships of many kinds occur often in integrated systems. Software design methods should therefore accommodate such changes without significant, unnecessary complexity.

2.2.5 Removing a Constituent Behavior

Yet another common kind of change to integrated behaviors is the removal of a constituent behavior. In the automotive example, the lock subsystem was removed because it was too expensive. Removal of an imperfect or old tool from a programming environment (perhaps to replace it with a new tool) involves the same basic operation. Deletion of a graphical view at runtime is also similar, although it is a matter of a change during execution, not evolution.[2] This kind of change is abstracted in the final step of the evolutionary scenario, in which $B2$ is removed from the system, along with the relationship $\oplus$ that integrates it with $B1$. A software design method should not make it unnecessarily hard to accommodate this kind of change.

2.2.6 Non-Conservative Integration

I conclude this section by observing that all the behavioral relationships above are of a particular, restricted kind. All of the relationships *extend* given behaviors to make them work together, but the behaviors themselves are left undiminished. Each behavior remains present in its entirety within the integrated behavior. Behaviors are conserved through integration. Switch clients continue to treat the switches as switches even though the switches work together. The user of an editor integrated with a compiler uses it as an editor. I call behavioral relationships of this sort *conservative.*

A different kind of relationship occurs in the automotive system. When the ignition interlock is on, the ignition switch cannot be turned on. The behavior of the ignition switch is not conserved when it is integrated with the behavior of the ignition interlock switch. Rather, this relationship *restricts* the behavior of the ignition switch. The

[2]Indeed, the distinction between changes during execution and evolution blurs. The basic principles that I discuss in this work apply to both evolution and execution dynamics.

behavior of the ignition switch is not conserved. I call behavioral relationships of this kind *non-conservative.* This work focuses on behavioral integration in conservative relationships. I defer the discussion of non-conservative integration to Chapter 8, which evaluates this work.

2.3 The Scenario as a Benchmark

The evolutionary scenario presented above characterizes several important kinds of integration requirements and evolutionary change. The scenario will also serve as a benchmark for evaluating design methods—i.e., to help assess how well they accommodate evolving integration requirements. Common design methods make it is easy to design artifacts representing the separate behaviors, $B1$, $B2$, and $B3$. The question is how well do they do in allowing one to adapt the artifacts as the integration requirements change.

After discussing software structure in the next chapter, Chapter 4 characterizes common design methods and evaluates each one by assessing how well it handles this scenario. Each method is used to create artifacts for individual switches, and then to integrate these artifacts to create artifacts for the integrated behaviors. This process yields an *evolutionary family* of artifacts, one for each behavior in the scenario. The nodes of the scenario give behavioral specifications to be realized as imperative programs by each given design method. I then assess each method by evaluating the structure of the resulting evolutionary family of specification/implementation pairs.

Chapter 3

Modeling Software Structure

To help characterize design methods and their impact on the structures of integrated software systems, this chapter presents a framework for modeling important software structures in a language independent way. To model the structure of a real software system, one selects a model built using this framework the structure of which reflects that of the system. To avoid mathematical "hair," I do not define the framework formally. Most of the English definitions can be formalized, if desired. In some places, I am content just to sketch a formal framework.

The framework is designed to model three dimensions of software structure. The first is the modular structure of a representation, e.g., specification or implementation, in a context in which it used by clients. The second is the structure of the mapping between two representations, where one *realizes* the other (an implementation may realize a specification). The framework models software as being *multi-representational.* The third dimension models the structure of the mapping between versions of a multi-representational system, where one version is a direct evolutionary descendent of the other. For example, one might model the mapping from a text editor before its integration into a programming environment to the editor after integration.

3.1 Representation in Context

The first dimension of software structure to be modeled is the modular structure of an individual representation in a context in which modules of the representation are used by clients. I therefore define the type *representation in context* as a modeling construct. An instance *rc* of this type can be taken to model a real-world representation in context.

The attributes of this type provide a basis for modeling attributes of real software representations. The attributes are defined in terms of entities called *modules* and *references,* in terms of relations over modules and references, and in terms of axioms that govern configurations of these elements. The configurations model structures such as recursive module decomposition; ownership of references by modules; naming of modules by references; visibility of modules; direct links between modules where the behavior of one directly affects the behavior of the other; indirect behavioral links; and the presence of complex behavioral relationships (such as $\oplus$) between the behaviors of modules.

3.1.1 Modules

Modularity is a key property of software representations because it is essential for humans to manage complexity [Dijkstra 65, Dijkstra 72]. Even if the clients of a representation view it as a monolithic module, it will almost invariably be implemented internally as a structured set of modular parts. To support modeling of modular structure I define the *module* as a basic type in the modeling framework; and I define a representation in context *rc* as having a set of modules *rc.modules* as an attribute.

3.1.2 Representation versus Context

It is important to consider how satisfaction of integration requirements affects clients. To ignore such an impact is to ignore what can be a significant cost. Thus, I model the separation of a representation in context into representation and context. I partition *rc.modules* into two disjoint, exhaustive subsets: *rc.contextModules* and *rc.systemModules*. The latter set contains one *module* to model each (real-world) module in the represen-

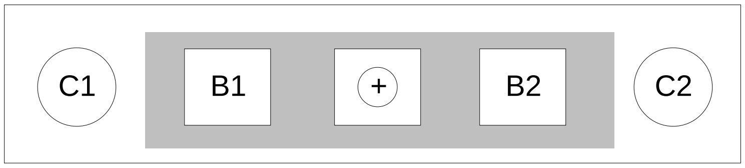


Figure 3.1: $B1 \oplus B2$ in its context of use.
This representation in context contains five modules. Shaded squares represent modules considered part of the system; unshaded circles represent clients in the context of use.

tation being modeled. The former set contains one *module* to model each client of the representation. Modeling clients as modules—even though they may be people, machines, etc.—eases the modeling task by permitting one to model associations between clients and representations in the same way as associations between modules of the representation itself. Disjointness and exhaustiveness of the subsets model the boundary between what is considered the system and its context.

To see how the framework as presented so far can be used to model systems, consider the model of the specification for the system $B1 \oplus B2$ (from Chapter 2) illustrated in Figure 3.1. Recall that $B1$ and $B2$ specify switches toggled by clients $C1$ and $C2$, and $\oplus$ specifies a behavioral relationship requiring $B2$ to be turned on when $B1$ is turned on. This system is modeled as a *representation in context rc* with *rc.clientModules*=$\{\ C1, C2\ \}$ and *rc.systemModules*=$\{\ B1, B2, \oplus\ \}$. The set *rc.modules* is the union of these sets by definition. $\oplus$ is modeled as a *module* because it is a distinct part of the specification. Another way to model behavioral relationships is provided below.

3.1.3 The Submodules Relation

Of course, systems are not generally organized as "flat" sets of modules. A key structure in which modules participate is recursive decomposition: one module may be a child or parent of another. I extend the framework to model this structure by defining the partial relation *rc.subModules: rc.modules* $\leftrightarrow$ *rc.modules,* to be an attribute of a representation in context *rc.* A pair (M, N) in this relation models a structure in which a real module modeled by M is a child of a real module modeled by N. I take *rc.parentModule* as the

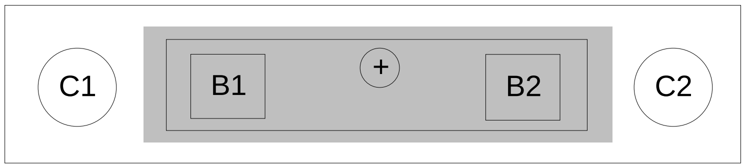


Figure 3.2: A representation with a non-trivial submodules relation. $B1$ and $B2$ are encapsulated submodules of the module $\oplus$.

relational inverse of *rc.subModules.* Making *rc.parentModule* a partial function models that decomposition is hierarchical (a module has at most one parent).

It is sometimes necessary to discuss a module and all of its submodules as a unit. I do this by defining *flatten* : *rc.modules* $\leftrightarrow \mathbb{P}$ *rc.modules* as a relation that takes a module M to the modules in the set $\{ M \} \cup rc.subModules^{+}(M)$, where $rc.subModules^{+}$ is the transitive closure of *rc.subModules*. To ensure that a submodule of a system or client module is itself a system or client module, I require that *rc.systemModules* and *rc.clientModules* be closed under *rc.subModules.*

Figure 3.2 illustrates these concepts with a model of a possible implementation of the behavior $B1 \oplus B2$. The nesting of the modules $B1$ and $B2$ inside $\oplus$ models the nesting of child modules $B1$ and $B2$ in a parent $\oplus$, with $\oplus$ responsible for coordinating the behaviors of $B1$ and $B2$. Thus, $rc.parentModule = \{ (B1, \oplus), (B2, \oplus) \}$; *parentModule* is a partial function; $flatten(\oplus) = \{ \oplus, B1, B2 \}$; and *rc.systemModules* (containing $B1, B2$, and $\oplus$) is closed under *subModules.*

3.1.4 References and the Touches Relation

Another feature of modular representations is the presence of references between modules. The familiar software engineering idea of coupling of modules [Stevens, Myers, and Constantine 74] is based on the idea of references between modules. To support modeling of the reference structures of systems I extend the framework in several steps.

First, I define *reference* as a type of modeling entity. An instance of this type is intended to model an actual reference (e.g., a name or address) embedded in one

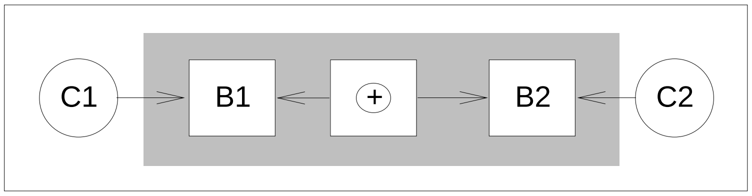


Figure 3.3: The reference relation over modules.
The arrows in this figure represent elements of the *references* relation. The boxes at the tails of the arrows represent modules containing references to the modules at the heads.

module that may denote some other module. Second, to a *representation in context, rc,* I associate a set of references *rc.references.* This set models all references appearing in all modules in *rc.Modules* Third, to model that any given *reference* is owned by one module and that the reference may denote another module, I define two functions. The total function *rc.referenceOwner: rc.references* $\rightarrow$ *rc.modules* associates every reference with an owning module. The partial relation *dereference: rc.references* $\nrightarrow$ *rc.modules* associates a reference with the module it names, if there is one. (An *unbound* reference does not denote any module.)

Finally, as a shorthand for modeling that one module names another by way of a reference, I define the partial relation *rc.touches* : *rc.modules* $\leftrightarrow$ *rc.modules* as an attribute of *rc,* with the requirement (or axiom) that for any two modules M and N in *rc.modules,* the pair (M, N) is in *rc.touches* if and only if there is a reference R in *rc.references* such that *rc.referenceOwner(R,M)* and *rc.dereference(R,N).* In other words, M touches N if and only if M has a reference R that refers to N. In this case I will also say that M references N.

To illustrate, Figure 3.3 depicts a model of the specification $B1 \oplus B2$. The clients are modeled as referencing the switches. This is justified because the client specifications refer to the switch specifications: they state that the clients check and toggle the switches (see Chapter 2). The switch definitions do not reference each other, the clients, or $\oplus$. Finally, $\oplus$ references $B1$ and $B2$ because the specification of $\oplus$ refers to the specifications of $B1$ and $B2$.

3.1.5 The Affects Relation

The next structure that I model is that of direct behavioral connections between modules. The execution behavior of a module M can directly affect that of another module N in several ways. M could write values to N or call procedures of N, or N could read values from M or register its operations to be invoked by events of M, for example. In this work, I address synchronous connections, where a behavior occurring in execution of one module synchronously invokes a execution in the another.

In the imperative implementation frameworks I consider, such behavioral connections result from procedure calls or event notifications. To model these semantic connections, I define the relation *rc.affects: rc.modules* $\leftrightarrow$ *rc.modules.* Modeling indirect connections—where behavior propagates through chains of modules—is done in terms of the transitive closure of *rc.affects*, denoted by *rc.affects*$^+$. Although the *affects* relation for a software representation is generally not computable from the representation, it is nevertheless a useful tool for modeling how behavior does or should propagate through systems.

Note that behavioral connections modeled by elements of *affects* are not the same as behavioral relationships such as $\otimes$ as discussed above. Behavioral relationships integrate behaviors; the *affects* relation relate modules. A connection between behavioral relationships and *affects* does occurs when some behaviors to be integrated are themselves represented as modules. The modules will have to affect each other directly or indirectly in order to work together. This connection is discussed in more detail below.

Figure 3.4 illustrates a model of the *affects* and *touches* relations in a possible implementation of the system $B1 \otimes B3$ (in which toggling either switch toggles the other.) Clients are modeled as affecting switches because clients turn the switches on and off. The switches affect the clients because clients read switch states to decide which way to toggle them. Switches affect the relationship module $\otimes$ because the activity of a switch module may require the $\oplus$ module to act to maintain consistency. The relationship module affects the switches because consistency maintenance involves toggling them.

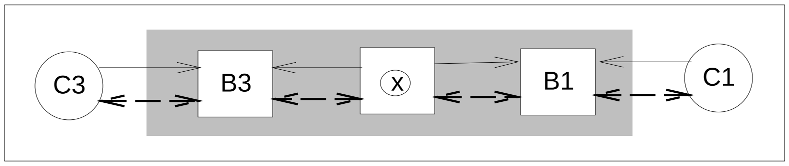


Figure 3.4: The *touches* and *affects* relationships for $B1 \otimes B3$.
The solid arrows represent elements of the *touches* relation; dashed arrows, elements of *affects*.

3.1.6 References versus Affects

What is the connection between the *affects* relation and the *touches* relation? In practice, modules affect each other in the ways described above—e.g., by calling or registering with each other's operations and events. These particular connections require that modules reference each other. For one module to affect another by calling it, the first must reference (i.e., *touch*) the second. For a module N to be affected by another M by being registered with M, N must reference M. This gives the connection we seek: if module M *affects* N, then M *touches* N or N *touches* M. In other words, *affects* $\subseteq$ (*touches* $\cup$ *touches*$^{-1}$).

This relation is important in this work. It characterizes the syntactic structures needed to obtain a given semantic structure. It also provides a basis for distinguishing an important design subspace—that of classical object-oriented (OO) methods. The key property of these methods is that objects interact by only explicitly invoking each other. Classical OO methods "constrain away" implicit invocation as a way to obtain *affects* relations, and so require *touches* relation wherever *affects* relations are required. This restricted subspace is modeled by strengthening the axiom given above to state that *affects* $\subseteq$ *references.*

Choosing classical object-orientation as an implementation framework seriously constrains the reference structures of behaviorally integrated system. Under such constraints, encapsulation and hardwiring are entirely reasonable design methods. This restriction unduly constrains the software designer.

3.1.7 The Visibility Relation

The next structure to be modeled is visibility. Visibility defines what references are permitted, as opposed to those that actually occur. I model visibility as a relation *rc.visibility: rc.modules* $\leftrightarrow$ *rc.modules* along with an axiom that relates *visibility* to the *touches* relation: if M does not have visibility to N, then M cannot *touch* N. More concisely, *touches* $\subseteq$ *visibility*.

Combining this axiom with the earlier one gives $\mathit{affects}^{+} \subseteq (\mathit{visibility} \cup \mathit{visibility}^{-1})^{+}$. The contrapositive then says if modules (M, N) are not in $(\mathit{visibility} \cup \mathit{visibility}^{-1})^{+}$, then (M, N) is not in $\mathit{affects}^{+}$. By restricting references, visibility thus indirectly restricts *affects*. Indeed, a primary use for control of visibility is to help avoid unwanted behavioral connections. In the encapsulation design method (discussed in Chapter 1 and next chapter), visibility of external modules to modules encapsulated by an integrating wrapper is denied. Combining this with the classsical object-oriented restriction to explicit invocation ensures that external modules do not affect the encapsulated modules directly (which would bypass integration code in the wrapper module).

3.1.8 The Integrated Relation

This section has presented in a bottom-up manner a framework for modeling representations. This subsection presents the final element of this part of the framework: a relation over modules called *integrated*, for modeling requirements for the integration of the behaviors of modules, such as those given by the $\oplus$ behavioral relationship.

I define *rc.integrated* : *rc.modules* $\leftrightarrow$ *rc.modules* as an attribute of a representation in context *rc*, the elements of which are intended to model requirements for the integration of the behaviors of a pair of modules.[1] If the independent switch behaviors $B1$ and $B2$ are modeled as modules $b1$ and $b2$, then the behavioral relationship $\oplus$ could be modeled as an element $(b1, b2)$ of the *integrated* relation, in contrast to the modeling of this relationship as a module, as was done earlier in this chapter.

[1] Chapter 8 discusses the artifici¡al restriction of *integrated* to a binary relation.

The *integrated* relation provides an abstract way to model representations organized as behavioral ER models: i.e., as sets of behaviors (the modules) connected in a network of behavioral relationships (the elements of the *integrated* relation). Thus the framework provides abstract concepts useful for thinking about and expressing the structures of specifications of integrated systems given in the form of behavioral ER models.

3.1.9 Refinement of Integrated Relations

By defining axioms that relate the *integrated* relation to "lower-level" relations already defined (e.g., $\mathit{affects}^+$) I provide a framework for characterizing and exploring the lower-level structures that are compatible with a given behavioral ER model, i.e. to investigate designs consistent with requirements for integration of the behaviors of given modules.

In this work, I consider it necessary for the behavioral integration of two modules that causing one module to engage in some behavior causes the other one to engage in some behavior, too. I thus incorporate the *integrated* relation into the framework by defining an axiom that captures this idea. The integration of modules M and N means that any module that affects M (directly or indirectly) also affects N (directly or indirectly). The requirement that a compiler update object code whenever an editor saves a source code file illustrates the idea: if any client causes the editor behavior, then a compiler behavior is also implied.

One way to meet such a requirement is to require editor clients to run the compiler whenever they use the editor to save a source file. Alternately, one could have the editor "trigger" recompilation by the compiler without any effort by the editor client beyond using the editor. In both cases, the editor client directly or indirectly affects the compiler, satisfying the requirement for integration.

I model this choice with the axiom that, for a representation in context rc and modules M and N, (M, N) being in $rc.\mathit{integrated}$ implies that, for at least one of M or N (say M), for every module C in $rc.\mathit{modules}$ such that (C, M) is in $r.\mathit{affects}^+$ (C, N) is also in $r.\mathit{affects}^+$. Figure 3.5 illustrates.

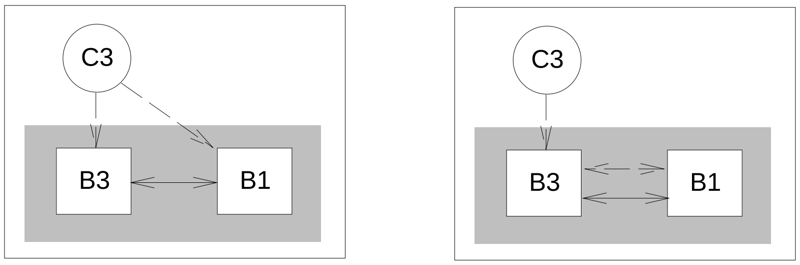


Figure 3.5: The *affects*$^+$ and *integrated* relations for $B1 \otimes B3$.
In this figure, the dashed arrows represent elements of *affects*$^+$, the transitive closure of the *affects* relation. The solid arrows represent elements of *integrated*. This figure illustrate the implications for the *affects*$^+$ relation of the integrated relation. $B3$ and $B1$ being in the integrated relation implies either that the client $C3$ integrates $B1$ and $B3$ by affecting both of them directly or indirectly (left); or that the integration of $B1$ and $B3$ is realized by the switch modules directly or indirectly affecting each other.

A consequence of this axiom is that $\mathit{integrated} \subseteq (\mathit{affects} \cup \mathit{affects}^{-1})^+$. Combining this with previous results gives $\mathit{integrated} \subseteq (\mathit{affects} \cup \mathit{affects}^{-1})^+ \subseteq (\mathit{touches} \cup \mathit{touches}^{-1})^+ \subseteq (\mathit{visibility} \cup \mathit{visibility}^{-1})^+$. That is, integration requires affects requires references requires visibility; but many design choices are possible for any given integration requirement.

3.2 Realization

In practice, software systems are *multi-representational.* A system comprises several representations of a desired behavior, each one suited to a particular purpose. A specification may support human reasoning and communication by being written in an expressive notation and by avoiding commitments to implementation decisions. An implementation that *realizes* the specification may embody commitments to implementation decisions and may be cast in a less expressive but more easily executed notation. In structuring complex software systems, engineers must consider multiple representations—both individually and collectively—including structural correspondences between them induced by the need for one to *realize* another. This section extends the modeling framework to model systems consisting of multiple, related representations.

3.2.1 Background

The idea of software as multi-representational and the existence of correspondences between the structures of adjacent representations have long been recognized. Boehm [Boehm 88, p.251] implicitly traces the idea to Bennington's stagewise process model [Bennington 56], an early view of software development as a sequence of transformations from higher to lower level representations. Lehman [Lehman 81, p. 474] finds the same idea in the transformational model of Zurcher and Randell [Zurcher and Randell 68]. The idea appears again in the waterfall model [Royce 70]. The invariant across all these process models (and many others) is the view of software as multi-representational, with correspondences between representations. Lehman makes the idea of structural correspondence explicit in discussing formal transformational development processes:

> Each step ... transforms the structure and notation of the input model from a form suitable for conveying understanding about the previous step of the process into a *framework* in which, for the current step, the design decisions and model refinements may be made and precisely, retrievably and understandably expressed. If the source and object models of this transformation are both formally described and if a verified mechanical transformation is used, the precise correspondence between the representations may be guaranteed [Lehman 81, pp. 483–484].

Even in the absence of formal transformations, the existence and importance of the structures of correspondences between representations is widely accepted. Jackson [Jackson 75] emphasizes that a close correspondence between problem and solution structure is key to reducing the difficulty of evolution; Meyer [Meyer 88], DeChampeaux [de Champeaux, Lea and Faure 93], and many others echo similar concerns.

3.2.2 Specification.

To begin, I take behavioral ER models as high-level representations. These representations are abstract in the sense that they are not intended to specify the low-level structure of an implementation, but only to record what behaviors and behavioral relationships are to be realized. To emphasize their abstractness, I model behavioral ER models only in terms of the high-level aspects of the framework. I model constituent behaviors as modules; and behavioral relationships, as elements of the *integrated* relation. Behavioral ER models, in this view, do not specify the *affects*$^+$, *affects*, or *references* structures required of an implementation.

3.2.3 Implementation.

A low-level representation, such as a C++ program that realizes a behavioral ER model, binds decisions about low-level structures. The program embodies choices about the *affects*$^+$ relation (whether clients realize integration requirements or whether they are relieved of this burden); how *affects*$^+$ relations are realized as chains of *affects* relations (e.g., whether intermediate modules are interposed); and how *affects* relations are implemented in terms of *touches* (whether implicit or explicit invocation is used).

3.2.4 Realization Mapping

I model that a software system comprises multiple representations with a new modeling type, *system*. A system s has several attributes. One is a representation in context $s.specification$; another is a representation in context, $s.implementation$. The third attribute of the system s is $s.realization$, a mapping from elements of the implementation to elements of the specification. I assume that $s.specification$ specifies modules and *integrated* relations but leaves lower-level structures (such as the *affects*$^+$ relation) unspecified; that $s.implementation$ specifies modules, *affects*$^+$, and the other lower-level structures; and that the realization mapping links the higher and lower level structures of the two representations. The rest of this section elaborates on the realization mapping.

The realization mapping is determined by a sequence of design decisions, some of which may be forced by the use of a particular implementation framework or design method. The first design decision comes in crossing the gap from the specification (which models a behavioral ER model) to the implementation (e.g., a C++ program). Each behavior in the behavioral ER model is modeled by a specification module. I assume that each such module is realized in a corresponding implementation module. The relationships in the behavioral ER model are modeled as elements of the *integrated* relation of the specification. The question then is how are these *integrated* relations realized in the implementation.

Consider an example. Suppose M and N are specification modules and that (M, N) is in the relation $s.\mathit{specification.integrated}$, and furthermore that m and n are implementation modules that realize M and N respectively—i.e., (m, M) and (n, N) are in $s.\mathit{realization}$. I require—as an axiom governing the system s—that m and n participate in the $s.\mathit{implementation.affects}^+$ relation in a way consistent with participation in $s.\mathit{implementation.integrated}$. Since m and n realize M and N, and M and N are integrated, we require that m and n be integrated.

Under the axioms of Section 3.1.9 (see Figure 3.5) there are two possible designs. Either every client c that $\mathit{affects}^+$ m also $\mathit{affects}^+$ n; or, alternatively, m $\mathit{affects}^+$ n. This choice allows the designer to make clients responsible for integration, or to define some other integrating structure between the modules themselves. The designer's choice between these alternatives determines a part of the mapping $i2a^+$, which associates the selected $\mathit{affects}^+$ relations with the given *integrated* relation. The top part of Figure 3.6 illustrates this ideas.

The next lower part of the figure takes us a step lower in the abstraction hierarchy: the $\mathit{affects}^+$ relation is realized in terms of the *affects* relation. That is, the actual chains of *affects* relations yielding the specified elements of the transitive closure are bound. Again, this design decision induces a correspondence between structures: each *affects* link specified to satisfy a given $\mathit{affects}^+$ link is associated to that link. I call this the a^+2a relation. The a^+2a mapping is not always so direct: an $\mathit{affects}^+$ relation could be realized by a chain of *affects* relations passing indirectly through several modules. In that case, each such module also associates to the given element of the $\mathit{affects}^+$ relation.

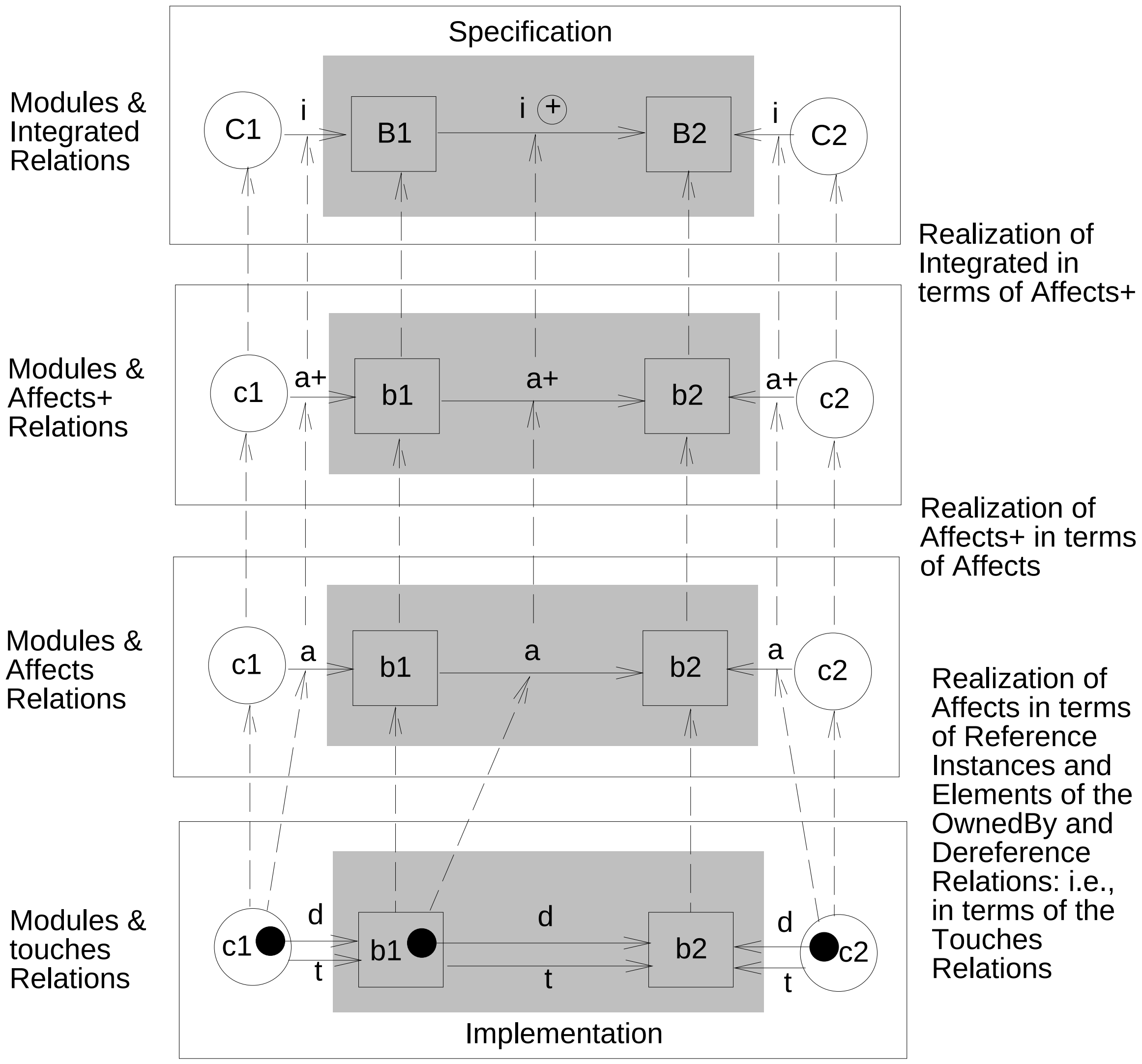

Specification
Modules & Integrated Relations
C1
i
B1
i ⊕
B2
i
C2
Realization of Integrated in terms of Affects+
Modules & Affects+ Relations
c1
a+
b1
a+
b2
a+
c2
Realization of Affects+ in terms of Affects
Modules & Affects Relations
c1
a
b1
a
b2
a
c2
Realization of Affects in terms of Reference Instances and Elements of the OwnedBy and Dereference Relations: i.e., in terms of the Touches Relations
Modules & touches Relations
c1
d
t
b1
d
t
b2
d
t
c2
Implementation

Figure 3.6: Realizing a multi-representation system implementing $B1 \oplus B2$. This figure "explodes" a model of a system to show its specification, implementation, and the design decisions that lead to the realization mapping between the two. The specification (behavioral ER model) is modeled in the top box. The i annotations indicate elements of the *integrated* relation. The next three boxes show the lower-level structures resulting from design decisions made by the implementor. The a^+ annotations indicate elements of the *affects*$^+$ relation; a annotations, elements of *affects*; the ts, elements of *touches*; dots, references; containment of dots in boxes, the *ownedBt* relation; and ds, elements of the *dereference* relation. The mappings between these relations are depicted by dashed arrows. The overall realization mapping from implementation to specification is the composition of these individual relations.

The final part of the figure models the realization of *affects* relations in terms of *touches* relations, each of which requires a reference, an element of the *ownedBy* relation (placing the reference in the referencing module), and a tuple in the *dereference* relation causing the reference to refer to the referenced module. The key constraint on the designer at this level is the need to choose implicit or explicit invocation—i.e., one of two orientations of *touches* to realize the given *affects* relation. (See Section 3.1.6.) I call the association induced by this decision $a2t$, since it reduces *affects* relations to *touches* relations. In an implementation framework based on classical object-orientation, the choice is often fixed: *affects* implies *touches*.

The relation mapping $s.realization$ between the specification and implementation is now essentially the composition of these mappings. Figure 3.7 illustrates this for the system above. An interesting point is that the choice of explicit invocation from $b1$ leads to a realization mapping in which $b1$ realizes the integrated relation $\oplus$. (This is the hard-wiring design method, discussed in Chapter 1 and in the next chapter.) An alternative choice of implicit invocation (as in the implicit invocation design method) would have put the reference in $b2$ and the realization arrow from $b2$ to $\oplus$ rather than from $b1$. This illustrates how design methods affect not only the structures of implementations but also those of the mappings that link implementations to the specifications that they realize.

3.3 Evolution

A different kind of structural correspondence arises between versions of a system: *evolution mappings.* Evolution mappings relate different versions of multi-representational systems to each other. As outlined above, each such system contains several representations connected by realization mappings. Evolution mappings have corresponding structures. In particular, an evolution mapping associates corresponding parts of systems before and after a change: the specification before a change to the specification after; the implementation before to the implementation after; and similarly for the realization mapping.

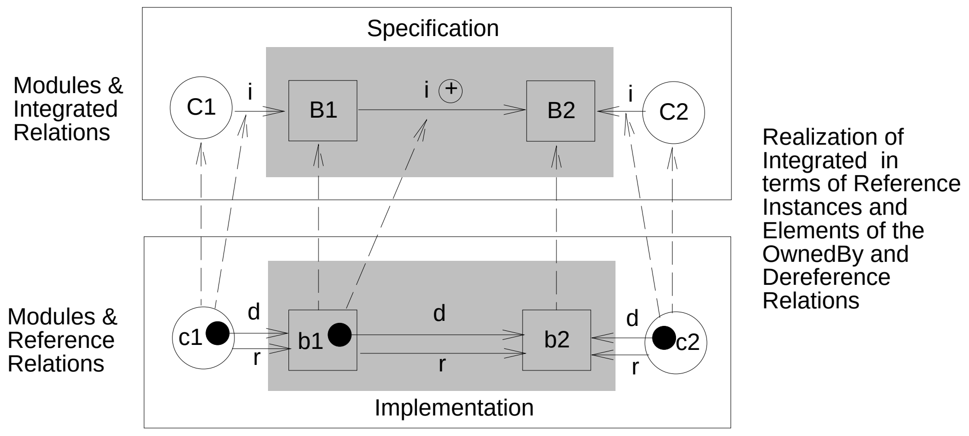

Specification
Modules & Integrated Relations
C1
i
B1
i ⊕
B2
i
C2
Realization of Integrated in terms of Reference Instances and Elements of the OwnedBy and Dereference Relations
Modules & Reference Relations
c1
d
r
b1
d
r
b2
d
r
c2
Implementation

Figure 3.7: Model of a multi-representation system implementing $B1 \oplus B2$. This figure depicts a model of a system comprising a multiple representations (depicted in the boxes) along with realization mappings that connect them together (depicted by dashed arrows between the boxes).

I focus on the constituent mappings from one representation to another, e.g., from the implementation before integration to the one after. The idea of such mappings is evident in the definition of the evolutionary scenario in Chapter 2. Edges in the evolutionary scenario graph basically denote evolution mappings between adjacent behavioral ER models.

Because representations themselves have structures (modules, references, various relations) evolution mappings between representations also have finer, internal structures. This section sketches a model of this aspect of the overall evolution mapping. Figure 3.8 depicts a simple example detailing the correspondence between the structures of a possible implementation of the behavior $B1 \oplus B2$ before and after the integration of the two switches by the $\oplus$ relationship. The horizontal arrows in the figure denote elements of the *touches* relation. The upward arrows depict the structure of the evolution mapping between the two representations.

In this case, the structure of the implementation before the integration step is directly embedded into the structure of the implementation after. Indeed, because it is often cheaper to modify representations when changes are required than to reimplement them,

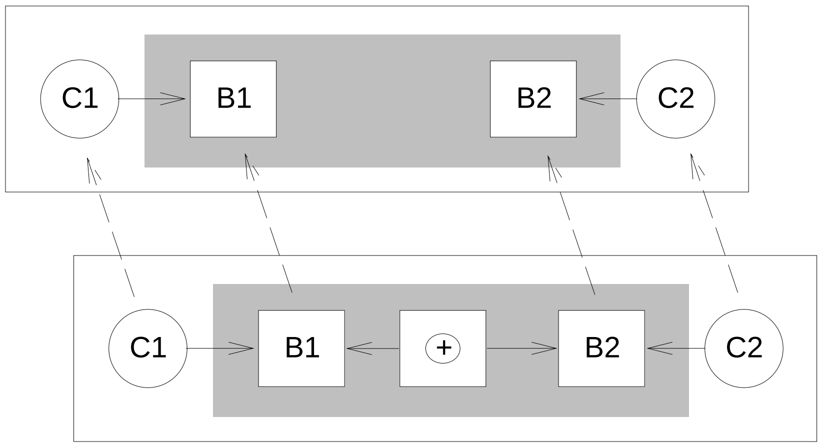


Figure 3.8: Evolution morphism with respect to the references relation. This figure depicts evolution involving integration of $B1$ and $B2$ by the addition of $\oplus$ in the descendant representation.

we expect strong correspondences of this sort to arise frequently. Parts of the original implementation are used without change in the new version. Other cases may require parts to be added, changed, or deleted.

I extend the modeling framework to model evolution mappings between representations as *evolution morphisms.* An evolution morphism (E, R) defines an *evolution relation* E that maps modules of one representation (the *ancestor*) to the modules of another (the *descendent*), and an association R between corresponding relations within the two representations. For example, R associates the *touches* relation before evolution with the *touches* relations after.

This model provides a basis for comparing modular structures before and after a given change. One might ask whether a module after an integration step is the same as the module before? This requires a definition of *same.* For the purposes of this work, I consider two modules to be the same if they hold the same references. This definition focus attention on the referential independence of modules. Another question is how corresponding relations over modules relate to each other before and after some change. Morphisms are useful constructs for capturing such concerns. In particular, if the mapping from modules before a change to those after is bijective, then one asks whether the morphism is an isomorphism. Isomorphism is the analog of *same* for relations in this framework.

When integration of two representations is the change being considered, one may seek not isomorphism but embedding—or isomorphism with respect to a part of the integrated system. Making these ideas precise is important; but it is beyond the scope of this work. I will not be any more rigorous at this point.

Rather, I extend the modeling framework to model evolutionary families of multi-representational systems as *evolution graphs* having *systems* at the vertices and *evolution mappings* on the edges. Figure 3.9 illustrates this idea. This construct delineates the final view of software structure presented in this work. Structure extends in three dimensions: through evolutionary changes between systems, through realization within systems, and within representations of systems.

This model provides the basis on which I evaluate designs and design methods in the following chapters. In particular, for each of a range of design methods, I take the evolutionary scenario from Chapter 2 as defining a family of specification. For each member, I apply a given design method to produce an implementation and the corresponding realization mapping. I model these triples as systems; and I model the family of systems as an *evolution graph.* I then evaluate how well the design method did with respect to evolving integration requirements by evaluating the structure of this three dimensional model.

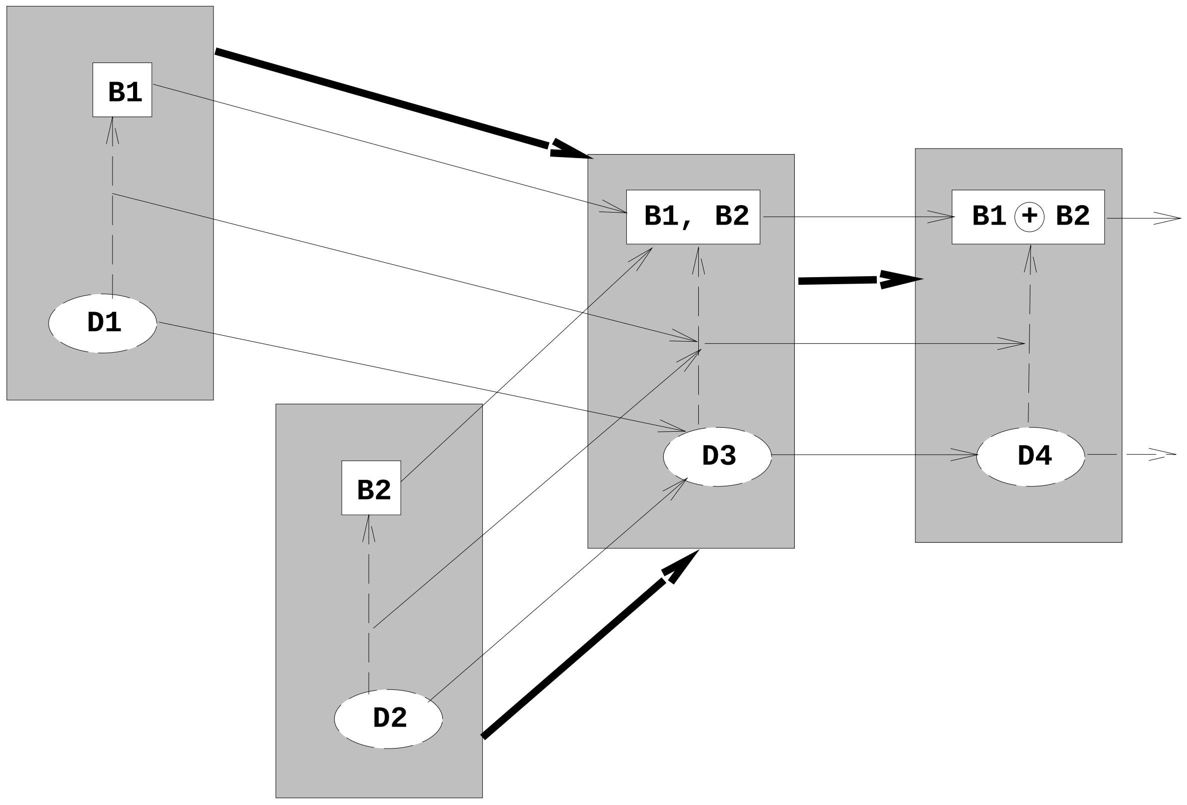


Figure 3.9: Systems are integrated to represent integrated behaviors.
The shaded rectangles represent systems. Within the systems are depicted behavioral ER models for part of the evolutionary scenario from Chapter 2. Ovals represent implementations that realize the specifications. The dashed, upward arrows denote realization mappings. Heavy horizontal arrows between systems denote evolution mappings between systems. Lighter arrows between components of systems represent those parts of evolution mappings that associate corresponding elements of systems. In the behavior $B1, B2$ both switches are present in the system but they do not work together. As the specification evolves and the switches are integrated, corresponding implementations may also be integrated to realize the integrated behaviors.

Chapter 4

Common Design Methods

This chapter models and evaluates five general software design methods that are used in building integrated systems. The first method I discuss provides a baseline by making clients responsible for integration. The real focus in this work is on the other four methods, which are widely used to design systems that relieve clients of the integration burden.

I view a design method as being characterized by a design space and by constraints imposed on the design decisions available to designers within the space. A design method is intended to help lead designers away from poor designs, toward good ones. A method that does this well is good; one that leads designers away from good designs is bad.

The methods studied in this work operate either in the design space modeled by the framework presented in Chapter 3 or in the classical object-oriented design space modeled by the framework without implicit invocation. This chapter characterizes each design method by identifying the space in which it operates and the design choices it institutionalizes. Specifically, different methods are characterized by different choices of *affects*$^+$, *affects* and *touches* for each given *integrated* relation. These choices significantly influence the static and dynamic structures of the resulting systems, and significantly affect the ease of realizing and changing such systems.

4.1 Introduction

Design methods employ certain design spaces and "institutionalize" certain design choices within those spaces. If a method constrains away a "good" part of the space, users of the method are methodically led away from good designs; so the method is poorer than one that does not constrain away the subspace. It would be quite bad if all common design methods constrained away a good part of the design space. Then, unnecessarily poor designs would be the norm. This and the next chapter argue that this is largely the case: the common methods that I address all unnecessarily complicate design, realization and evolution of integrated systems by constraining away mediator-based designs.

To support this claim, I identify and evaluate common design methods. As described at the end of Chapter 3, I evaluate each method in three steps. First, I apply a method to design a family of systems meeting the requirements of the evolutionary scenario of Chapter 2. Then I model each family in terms of the framework from Chapter 3. Finally, I evaluate these models in relation to accepted software engineering criteria, such as independence and separation of concerns. Using a simple, abstract evolutionary scenario as the basis for this exploration clarifies how the methods systematically complicates software engineering tasks.

The rest of this chapter addresses five common methods. The first is integration by clients. The remaining four methods relieve clients of the integration burden. A key feature of the first two of these methods—*encapsulation* and *hardwiring*—is that they use design spaces that "constrain away" designs that use implicit invocation. The second two methods—the *implicit invocation* and the *broadcast message server* (BMS) methods—allow implicit invocation.

This chapter characterizes these methods and shows how they complicate design. The next chapter shows that the complications are unnecessary by exhibiting a method—the mediator method—that uses the same design space but leads to designs that are significantly better with respect to ease of design, integration and evolution. The mediator method reveals an important but neglected subspace of integrated system designs.

4.2 Integration By Clients

One way to meet requirements for the integration of given software systems is to place responsibility for integration on the clients of the systems. In the system of switches $B3 \otimes B1 \oplus B2$, one could require that when client $C1$ turns on switch $B1$ it also turn on $B2$ and $B3$. This pattern arises often in practice. In a poorly integrated programming environment, for example, the programmer may have to run the compiler manually after saving changes to a source file.

An advantage of this method is that it permits the behaviors of given software systems to be integrated without changes to those systems. If $C1$, $C2$, and $C3$ coordinate the activities of $B1, B2$ and $B3$, the switches do not have to be changed. In a programming environment, neither the editor or compiler have to be changed if the user coordinates their activities.

Unfortunately, this method reduces the value of systems to users. In may even impose unacceptable burdens on them [Taylor 88, Habermann and Notkin 86]. Manually coordinating tools in a programming environment is tedious and error-prone, and distracts users from building software. Manually updating graphical views as computerized models change and manually translating and transferring data between tools in computer-aided design (CAD) systems may also unduly burden users. When the scope and complexity of systems grow, manual integration becomes untenable.

Figure 4.1 shows how this method and its static and dynamic structural consequences can be seen using the modeling framework. The figure shows three steps in the evolution of the system $B3 \otimes B1 \oplus B2$. Before integration, the clients of the switch implementations (lower half) need only reference their respective switches. The first integration step requires client $c1$ to realize the requirement for the integration of $b1$ and $b2$ by affecting $b2$. That in turn is achieved by having the (now changed) client $c1'$ explicitly invoke $b2.\mathit{TurnOn}$, which leads to $c1'$ referencing $b2$. After the second integration step, the client (now $c1''$) realizes both behavioral relationships, coordinating (and referencing) $b2$ and $b3$ in addition to $b1$. This method scales poorly from the client's point of view.

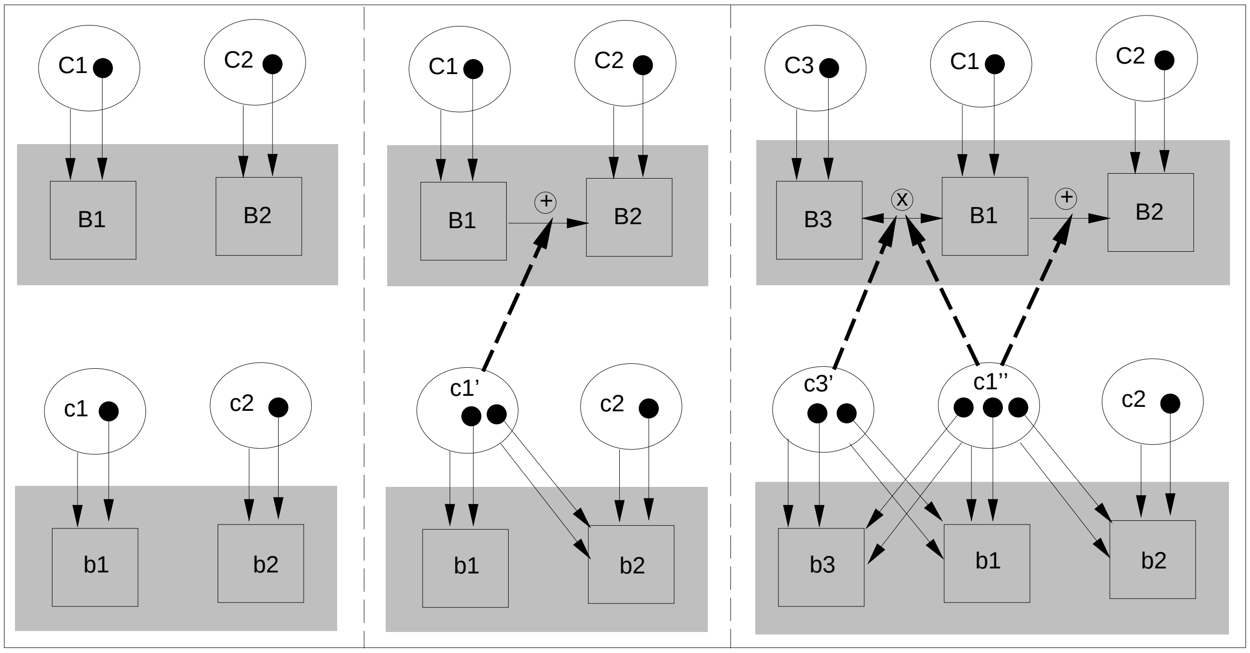


Figure 4.1: An evolutionary family with clients responsible for integration.
The panels, from left to right, model three points in a system's evolution. The top parts model specifications from the scenario of Chapter 2 before the first integration step, after the first, and after the second. The bottom parts of the figure model implementations. The black dots denote references. The arrows coming from them represent elements of the *touches* relation. The other light arrows (from clients to switches in this case) depict elements of *affects* (and thus also *affects*$^+$. The dashed arrows from the implementation modules to specification elements indicate divergences of the *realize* relation from a direct correspondence. One such arrow indicates a divergence in the system at the second step: the client $c1'$ realizes something other than $C1$, namely $\oplus$. Prime marks (') denote changes to modules made between versions.

4.3 Encapsulation

In contrast to the client-centered design method of the previous section, the methods discussed in the rest of the chapter relieve clients from the burden of integration.

This section addresses the encapsulation method, which largely relieves clients of this burden while retaining a key benefit of the client-centered method: the ability to integrate existing implementations without change. Unfortunately, as discussed below, this method does not solve the problem of *clients* having to change, and it leads to design structures that unnecessarily complicate integration and evolution.

The encapsulation design method has its roots in the theory of hierarchical composition of abstract data types [Liskov and Zilles 75]. The strategy is to implement integrated systems using given systems as hidden implementation components. One integrates the behaviors of given modules by interposing *wrapper* modules between modules and their clients. Responsibility for integration is then placed with the wrapper module. The clients of the modules that were integrated are affected in that they can no longer access the modules directly, but must use interfaces provided by wrapper modules.

A wrapper makes the behaviors of the modules it encapsulates accessible to clients that require those behaviors through its own wrapper interface. The implementation of this interface exposes the required behaviors and also ensures the integration of those behaviors. To integrate the editor and compiler, one would define a wrapper that encapsulates them, exports operations providing access to editing and compiling functions, and defines the exported operations as taking any additional actions needed to ensure proper integration. When a client calls the wrapper to save a source file, the wrapper first calls the *save* operation of the encapsulated editor, but then calls the encapsulated compiler to regenerate the object code. The behaviors of the encapsulated components are extended as needed for integration by the operations of the wrapper module.

Figure 4.2 assesses the performance of this design method in the face of evolving integration requirements by using part of the evolutionary scenario as a benchmark. The first integration step results in the switch implementations $b1$ and $b2$ being integrated

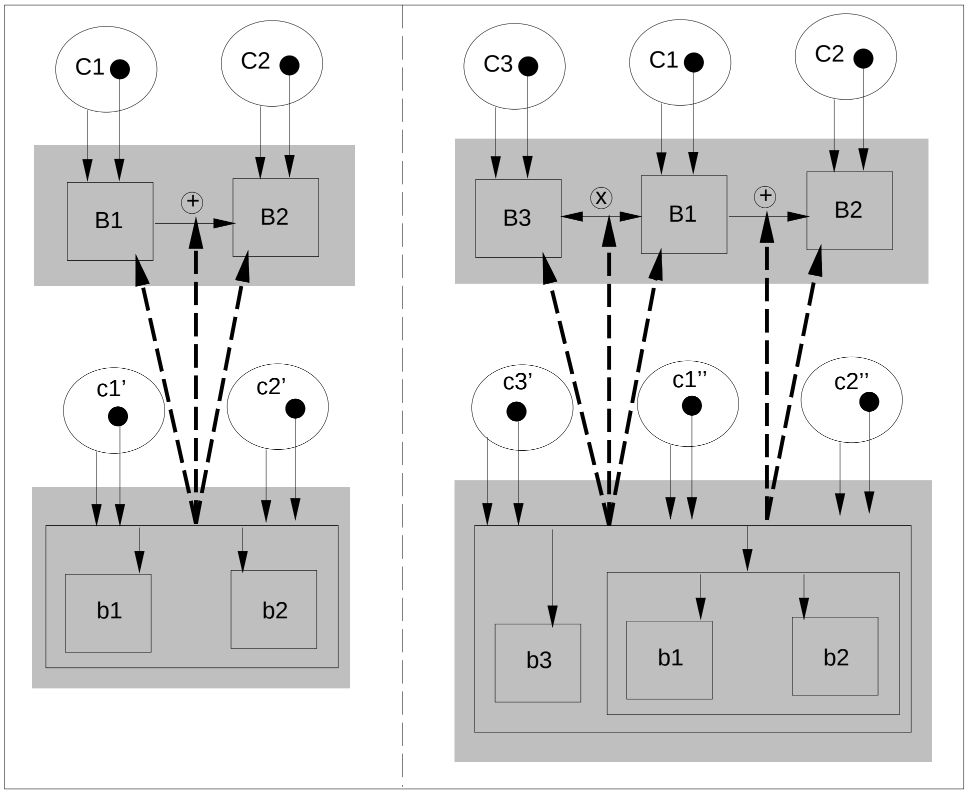


Figure 4.2: Model of an evolutionary family designed using the encapsulation method. The interpretation of the elements of this figure is as in Figure 4.1. The nesting of boxes depicts hierarchical encapsulation of modules: inner ones are parts of enclosing ones, and not visible to modules further out in the nesting structure.

by a new wrapper, which $c1$ and $c2$ have been changed to use. The second step is accomplished by introducing a second wrapper—one that integrates the third switch with the wrapper that integrates the first two switches. The clients $c1, c2$, and $c3$ have been changed to use the new wrapper. Now, consider what happens when $c1''$ asks the second wrapper to turn on $b1$: the second wrapper asks the first wrapper to turn on $b1$, which turns on both $b1$ and $b2$; then the second wrapper turns on $b3$. The integration requirements are thus satisfied.

This approach has at least three serious problems. First, each integration step requires clients to be changed to use a new wrapper module. Second, the resulting systems have monolithic structures. This is reflected in the realization mapping, which shows how a single wrapper module realizes every aspect of each specification. An engineer who wants to find aspects of the implementation that realize a given part of the specification is led to the outermost wrapper in which all parts of the specification are realized. In the reverse direction, a given part of the implementation corresponds to several, conceptually separate parts of the specification. Thus the outer wrapper in this case contains code related to both $\otimes$ and $\oplus$ (the call to the inner wrapper).

Third, this method has the property that the system structure after a sequence of integration steps depends on the arbitrary order in which the steps were taken, not only on the conceptual structure of the requirements. If we had integrated $B3$ with $B1$ before integrating $B1$ with $B2$, we would have a different nesting with different evolutionary properties. The structure here, for example eases removal of $\otimes$ but not $\oplus$. The alternative structure is the opposite in this regard. The key software engineering property of ease of evolution should not depend on arbitrary assembly order. The encapsulation design method does not handle evolving integration requirements very gracefully.

4.4 Hardwiring

In contrast to the basis of the encapsulation method in hierarchical composition of abstract data type interfaces, the hardwiring design method is rooted in classical object-

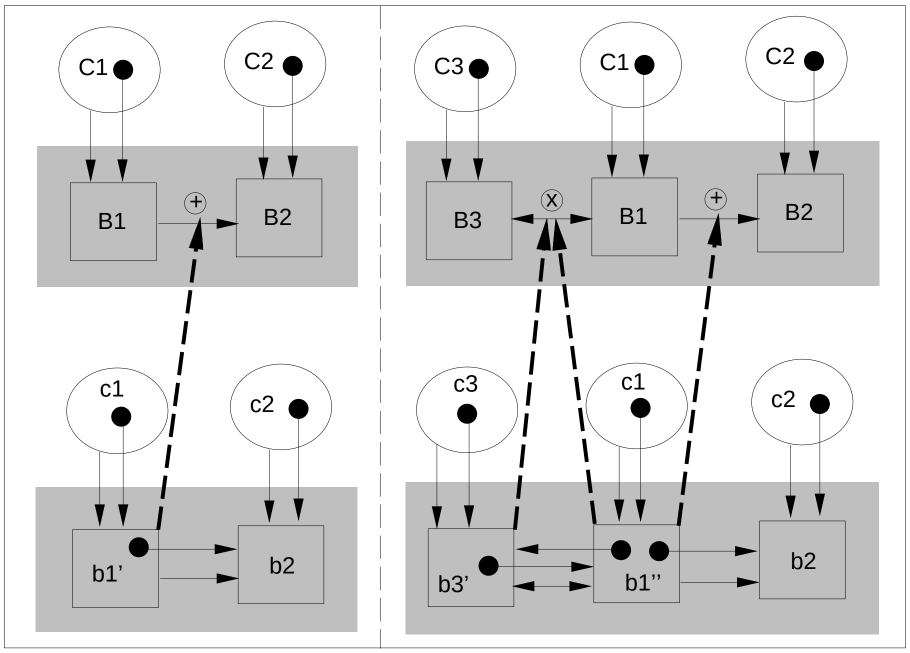


Figure 4.3: Model of an evolutionary family using the hardwiring design method. The first panel shows the structure of the system after the first integration step. The second panel shows the system after the second integration step.

based and object-oriented programming, a style in which systems are not organized as encapsulating trees of abstract data type instances, but rather as networks of exposed objects that invoke each other by explicit procedure call. Explicit invocations tie the implementations of the objects into behaviorally integrated systems. To integrate the editor and the compiler source codes using this method, one changes the editor source code by embedding a reference to the compiler and a call to its compile operation. The resulting, intertwined pair of source codes is the source code for the new, integrated behavior.

Figure 4.3 illustrates this approach in a model of part of the evolutionary family from Chapter 2. In the implementation of the system version on the right (lower part of the figure), when $c3$ calls $b3'$ to turn $b3'$ on, $b3'$ calls $b1''$ to turn it on. So, at this evolutionary stage $b3'$ has a reference to $b1''$. Then $b1''$ calls $b2$ to turn it on, satisfying the specified integration requirements.

Some additional complexity, not depicted in the figure, will be needed to prevent an infinite cycle of calls between $b3'$ and $b1''$. In particular, some state is needed to keep track of whether a change has already been propagated. For example, when $b1''$ is turned on it might first query $b3'$ to determine if that switch needs to be turned on before actually trying to turn it on. The client and encapsulation approaches avoided this complexity by not creating *affects*$^+$ relations between switch objects; in those methods the sequencing of calls was factored out of the objects themselves, into the client or wrapper modules.

A key advantage of the hardwiring approach is that clients of modules to be integrated need not necessarily change. This is illustrated by the lack of changes to $c1, c2$ or $c3$ in the figure. Requirements for integration of the behaviors of given modules are realized in terms of the implementations of those modules, not in terms of their interfaces, as in the encapsulation approach. Thus, the clients, which depend only on interfaces, did not have to change.

Unfortunately, this design method has several serious problems with respect to evolving integration requirements. First, it merely transfers the need for change from the clients of the modules (as in the earlier methods) to the modules themselves. Second, the changes to these modules compromise their simplicity and independence. Hardwiring compromises simplicity by placing responsibility for integration within the modules to be integrated. This is shown (see Figure 4.3) in the realization mapping: $b1''$ realizes not only the switch behavior $B1$, but also both behavioral relationships. The situation is made even worse because the symmetric relationship $\otimes$ is realized in several implementation modules ($b3'$ and $b1''$). Hardwiring compromises independence by requiring that the modules to be integrated affect, hence call, and thus reference each other.

4.5 Implicit Invocation

The implicit invocation design method is similar in spirit to hardwiring, but it operates in a design space that admits implicit invocation as a communication mechanism. To integrate the editor and the compiler, one would change the compiler source code to

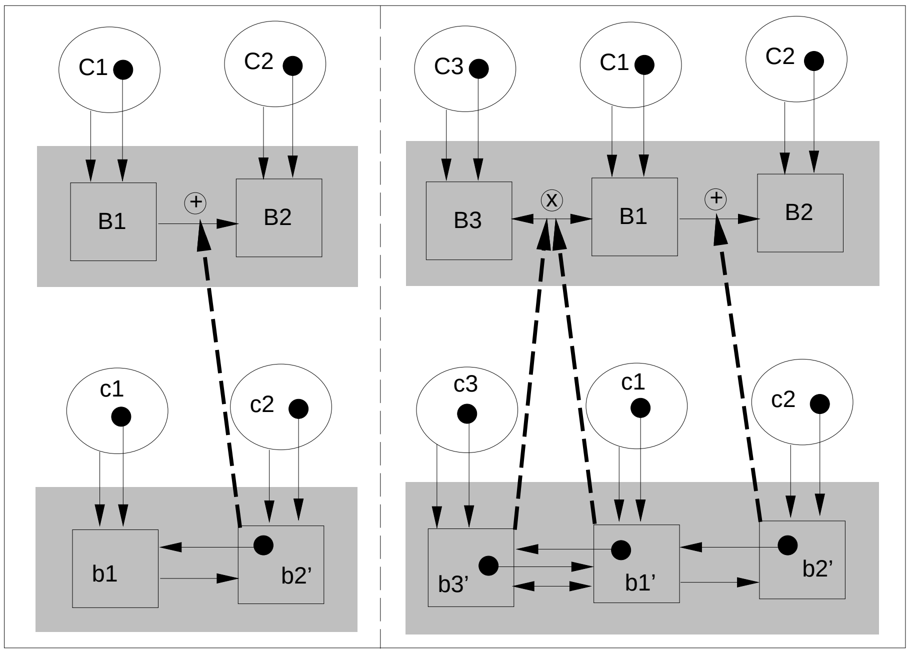


Figure 4.4: Model of an evolutionary family designed using implicit invocation.

monitor the editor for events that signal that it has saved a source code file. When notified of such an event, the compiler responds by recompiling the source code.

Many implicit invocation mechanisms have been built; and several important real design methods are based on implicit invocation. These methods include the Model-View-Controller method, supported by Smalltalk-80, which is often used for integrating graphical user interfaces with underlying object-oriented models. The diversity of implicit invocation mechanisms and the importance and widespread use of design methods based on them suggest there is something fundamental in implicit invocation.

This work provides one interpretation: implicit invocation is a dual to explicit invocation. Implicit invocation enables the implementation of an *affects* relation from M to a module N with a *touches* relation from N to M. A design space that supports both implicit and explicit invocation gives the designer a choice in the orientation of the *touches* relation needed for a given *affects* connection. That is the fundamental capability provided by of implicit invocation.

Unfortunately, the implicit invocation method does not avoid unnecessary complexity in the design, realization or evolution of integrated systems. Figure 4.4 illustrates. The first panel models a system of switches after integration step. The integration requirement implies an *affects* relation from $b1$ to $b2$. This need is met by changing $b2$ to $b2'$—into a module that registers with (hence references) $b1$. Now, $b2'$ will be notified when $b1$ is turned on. When so notified, $b2'$ turns itself on. This satisfies the requirement of $\oplus$. But this design is not clearly better than the hardwiring design. In general, the components to be integrated have to be changed, and in ways that compromise their simplicity and independence; nor are the behavioral relationships expressed in a form that allows them to be identified or changed, added, or removed as units.

4.6 Message Server

The broadcast message server (BMS) method was pioneered by Reiss in the FIELD project [Reiss 90]. Contributions of the project include the BMS mechanism, an associated design method, and a family of Unix-based [Ritchie and Thompson 78] integrated programming environments designed using this method. The FIELD method is now commercially supported, e.g., by Hewlett-Packard's Softbench [Cagan 90].

Fundamentally, the BMS method specializes the implicit invocation method by requiring communications between integrated modules to pass through a special component, the BMS. This relives the modules to be integrated (e.g., Unix tools) from having to invoke or reference each other directly. Indeed, visibility of tools to each other is denied by the mechanisms of the Unix operating systems on which the tools execute.

Figure 4.5 depicts a model of an evolutionary family of switch systems designed using this method. Each switch in the implementation references the BMS in order to affect it or to be affected by it. Switch $b1'$ references the BMS to explicitly invoke the BMS to indicate that $b1'$ has been turned on. Switch $b2'$ references the BMS to register to be implicitly invoked when the BMS receives such messages from $b1'$. When so notified, $b2'$ turns itself on.

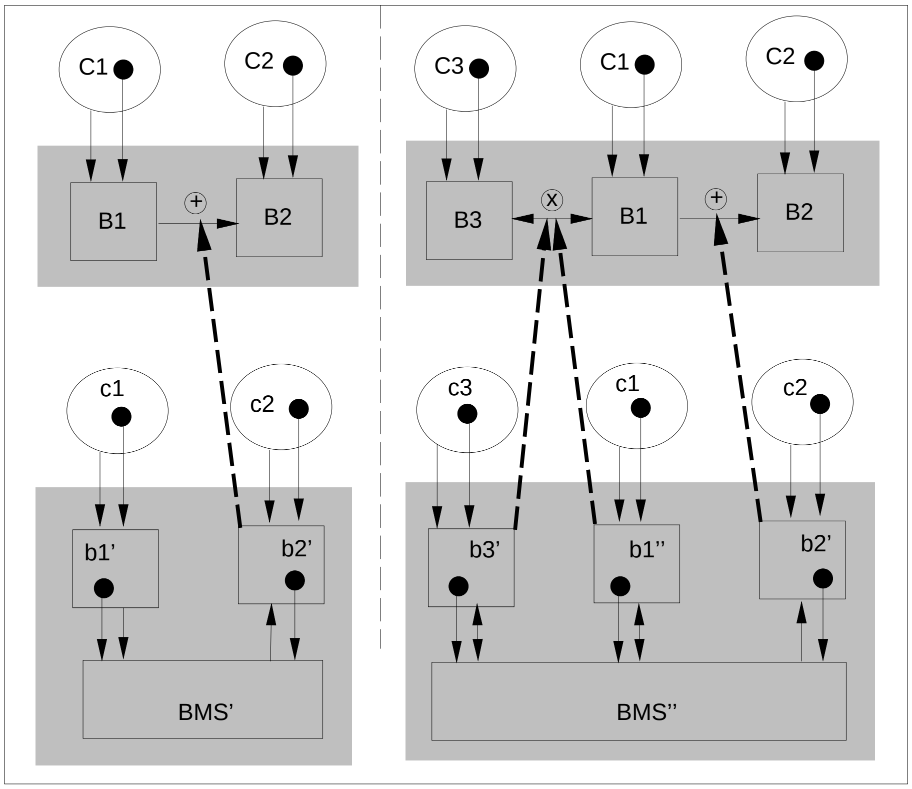


Figure 4.5: Model of an evolutionary family with broadcast message server designs.

An advantage of the BMS design method is that it relieves the modules to be integrated from having to reference each other. It accomplishes this through indirection: all communication passes through BMS. In practice, this is important for supporting substitutability of tools in integrated environments. Given an environment in which an editor and a compiler are integrated, it enables easy replacement of the either one as long as the expected editor and compiler protocols are upheld by the replacement tool.

Unfortunately, this method does not resolve the problems with the general implicit invocation style. The most obvious problem is that given modules have to be changed to be integrated. Changes to the switch implementations are already indicated (by prime marks) in the first version presented in the figure, reflecting that the switches had to be changed from their original states (not depicted) to be integrated. Similarly, editors and compilers have to be changed in practice to be integrated using the BMS method. The switch $b1'$ has to be changed a second time for the second integration step, to make it respond to notifications regarding activities of $b3$.

These changes also compromise the independence and simplicity of modules to be integrated. In terms of independence, modules come to depend on each other's interfaces and protocols (although not on identities). In terms of simplicity, components are complicated by the need to call or respond to notifications from each other, and in practice to manage other data and control related to integration. A compiler, for example, might be required to register with the BMS to be notified of events from an editor, and to respond to those notifications with "knowledge" that they come from an editor. Finally, as in the hardwiring and implicit invocation methods, the structures of the implementation and realization mapping both are skewed by the spreading of the realizations of behavioral relationships over the modules to be integrated. Behavioral relationships are not represented as units. Despite its popularity, the BMS approach creates costly structural complexities.

4.7 Synopsis

A wide range of common design methods, characterized in an abstract way in this chapter, complicate the design, realization and evolution of tightly integrated systems. These methods institutionalize design choices that lead to often inappropriate realizations of integration requirements. Newly imposed integration requirements are generally realized by changing the implementation modules that realize the behaviors to be integrated, or by changing the clients of these modules. These changes are not only costly in the present, but they compromise the simplicity and independence of the modules, raising the cost of future handling of the modules. On the other hand, behavioral relationships are not realized as separate modules, but as code and data sometimes spread over multiple implementation modules, or intermingled with code for other relationships in encapsulating wrapper modules. This complicates subsequent changes in behavioral relationships. In brief, common design methods significantly and unnecessarily raise the cost of realizing and evolving integrated systems—even the rate at which costs increase. The practical affect is that tightly integrated systems that are broad in scope and amenable to evolution tend not to be built; and clients are often left with the costly, uninteresting, and error-prone task of integrating behaviors themselves.

Chapter 5

The Mediator Method

This chapter presents the *mediator design method.* In comparison with common design methods, the mediator method eases integration and evolution while retaining key benefits of common methods. The foundation for the mediator method—behavioral ER models with their evolutionary properties—was laid in Chapters 2 and 3. We therefore already have a framework for thinking clearly about and designing integrated behaviors. This framework also provides a basis for structuring implementations that gracefully accommodate evolution of integration requirements.

The *mediator method* provides a way to implement behavioral ER models as object-based, imperative programs having static and dynamic structural properties that parallel those of behavioral ER models. The mediator method operates in the same design space as common design methods, and can be applied using many common programming languages and platforms, without complex, unfamiliar, or semantically restrictive mechanisms. The one possibly unfamiliar construct that is central to the method is the *abstract behavioral type;* but it just generalizes the ADT. The mediator method differs from common methods not in mechanisms, but in basing the design, implementation, and evolution of integrated systems on behavioral ER models.

5.1 Behavioral Entity-Relationship Modeling

Why base the architectures of integrated systems on behavioral ER models? One reason is that behavioral ER models usefully organize both the static and dynamic structures of integrated systems. The second reason is that behavioral ER modeling provides a framework that helps people think clearly about integrated behaviors—to analyze, conceive, and synthesize complex behaviors from simpler, independent behavioral constituents. This second claim, about ease of design, is empirical, in contrast to the first claim, about structure, which is analytic.

5.1.1 The Empirical Claim

I believe the empirical claim—that behavioral ER modeling eases design—for several reasons. One is experience. Behavioral ER modeling has helped me to think more clearly about integrated behaviors, and it has also helped others whom I know who have used it enough to become facile with it. (Chapters 6 and 7 describe the results of these experiences and discuss their validity as evidence supporting the empirical claim.) While I do not have scientific evidence, I can offer the following explanation for why I believe behavioral ER modeling eases design—conception in the mind.

Behavioral ER modeling helps by elevating behavioral relationships to the level of first-class semantic constructs, even very complex ones; and that provides a basis for the effective decomposition of complex behaviors into simpler (but still possibly quite complex), independent behaviors linked by rich behavioral relationships.

Endowing behavioral relationships with “semantic capacities” commensurate with those of the entities to be integrated provides a framework for thinking about complex integration semantics in separate, cognitive chunks. The enlarged capacity also provides enough “room” in relationships to relieve the behaviors to be integrated and their clients from managing integration. Making behavioral relationships first-class constructs promotes an effective separation of integration concerns.

5.1.2 The Analytic Claim

The rest of this chapter addresses the technical reasons for using behavioral ER modeling to architect integrated systems. The main advantage of behavioral ER modeling is that it organizes both the static and dynamic structures of integrated software systems in a useful way. The structures are those of graphs. At any given point in time, a behavioral ER model resolves a complex behavior into separate behaviors (vertices) integrated in a network of behavioral relationships (edges). A behavioral ER model organizes structural change based on operations natural for graphs.

That is, behavioral ER models evolve as graphs evolve. This dissertation focuses on the addition, deletion, and modification of behaviors and behavioral relationships as the operations by which behavioral ER models evolve. Integrating a new behavior is reduced to adding the behavior (a vertex) and relationships (edges) to integrate the behavior with others present in the system. Changing how behaviors are integrated is reduced to changing behavioral relationships separately from the behaviors themselves.

Behavioral ER models provide a basis for representing integrated systems in a way that is robust with respect to integration and evolution, as exemplified in the evolutionary scenario. Previous chapters show that common methods do not structure implementations this way. The rest of this chapter shows how the mediator method does. The end of this chapter and the next two show how the approach can be profitably used to design and implementing real-world systems.

5.2 Requirements for a New Design Method

This section asks what properties we want in an implementation method. The key requirement is to preserve the static and dynamic structural properties of behavioral ER models through realization of these models as imperative implementations. Then we obtain implementations that are robust with respect to evolving integration requirements in the same way as behavioral ER models.

5.2.1 Operate in Same Design Space

First, we have a strong preference for a design method that operates in the same design space as common design methods. We have invested heavily in these methods. Evolutionary enhancement is thus to be preferred to costly methodological revolution. If a satisfying design method can be found that operates in the same basic design space as commons methods, then, in the absence of other overriding concerns, that method should be used.

5.2.2 Relieve Clients of the Integration Burden

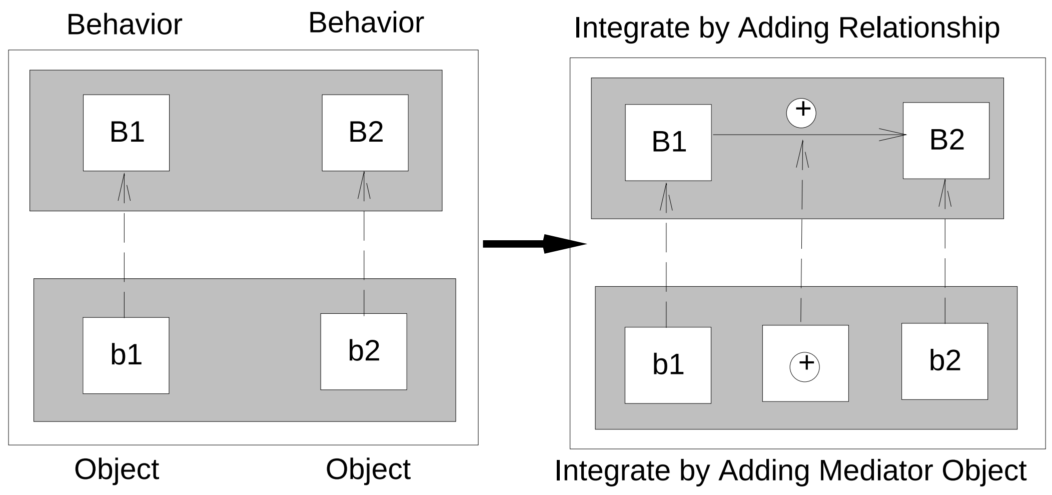
Next, we require that the new design method relieve clients of the burden of integration. This is often required in practice, since responsibility for integration unduly burdens clients. The requirement that object code be regenerated when source code is changed in a programming environment should not require users of the editor to run the compiler whenever they save a source file. By *integrated system* I mean a system in which integration is automatic. We want a method to build integrated systems.

5.2.3 Use Existing Representations Without Change

Another requirement is that the method should permit integration of the behaviors of given components without changes to those components. The alternative—that the components be modified—increases the immediate cost of integration and, by compromising structure, further increases the cost of future changes. Thus, so far, we want to be able to be able to integrate the behaviors of given components without changing either the components or their clients.

5.2.4 Represent Behavioral Relationships as Separate Modules

Since the way that behaviors are integrated changes independently of the behaviors themselves, we want to localize the implementations of behavioral relationships. We want to implement them as separate objects. When we put these requirements together,



Behavior
Behavior
B1
B2
b1
b2
Object
Object
Integrate by Adding Relationship
B1
+
B2
b1
+
b2
Integrate by Adding Mediator Object

Figure 5.1: The mediator method organizes the evolution of implementations. This figure illustrates one step, from left to right, in the evolutionary scenario. At the specification level (top half) $B1$ and $B2$ are integrated by the addition of the relationship $\oplus$. At the implementation level, this change is realized exactly by the addition of a corresponding module—a mediator.

we find that we want to be able to compose and decompose software implementations as behaviors are composed and decomposed in behavioral ER models. Since integrating behaviors in a behavioral ER model requires exactly the addition of behavioral relationships, so should it be possible to integrate the behaviors of given software representations just by adding objects representing the desired behavioral relationships.

Figure 5.1 illustrates this idea for a simple case. It presents two versions of the system $B1 \oplus B2$, before and after the integration of the switches. The key feature of the example is that the behavioral relationship $\oplus$ is realized by a separate implementation object. The next integration step, implementing the system $B3 \otimes B1 \oplus B2$ (which is not shown in the figure) would be handled in the same way, namely by adding another switch module, and another module for the relationship $\otimes$. This modular separation of integration concerns eases design, realization, and evolution of the integrated system.

5.3 Realizing Behavioral Relationships as Mediators

This section presents a method for realizing behavioral ER models that satisfies the preceding requirements. Consider the evolution of the preceding system from $B1, B2$ (where the switches are not integrated) into the system $B1 \oplus B2$ (where the switches interact). This prototypical example of evolution by integration is handled by adding the behavioral relationship $\oplus$ to the behavioral ER model. The issue addressed below is how this change is accommodated at the imperative implementation level.

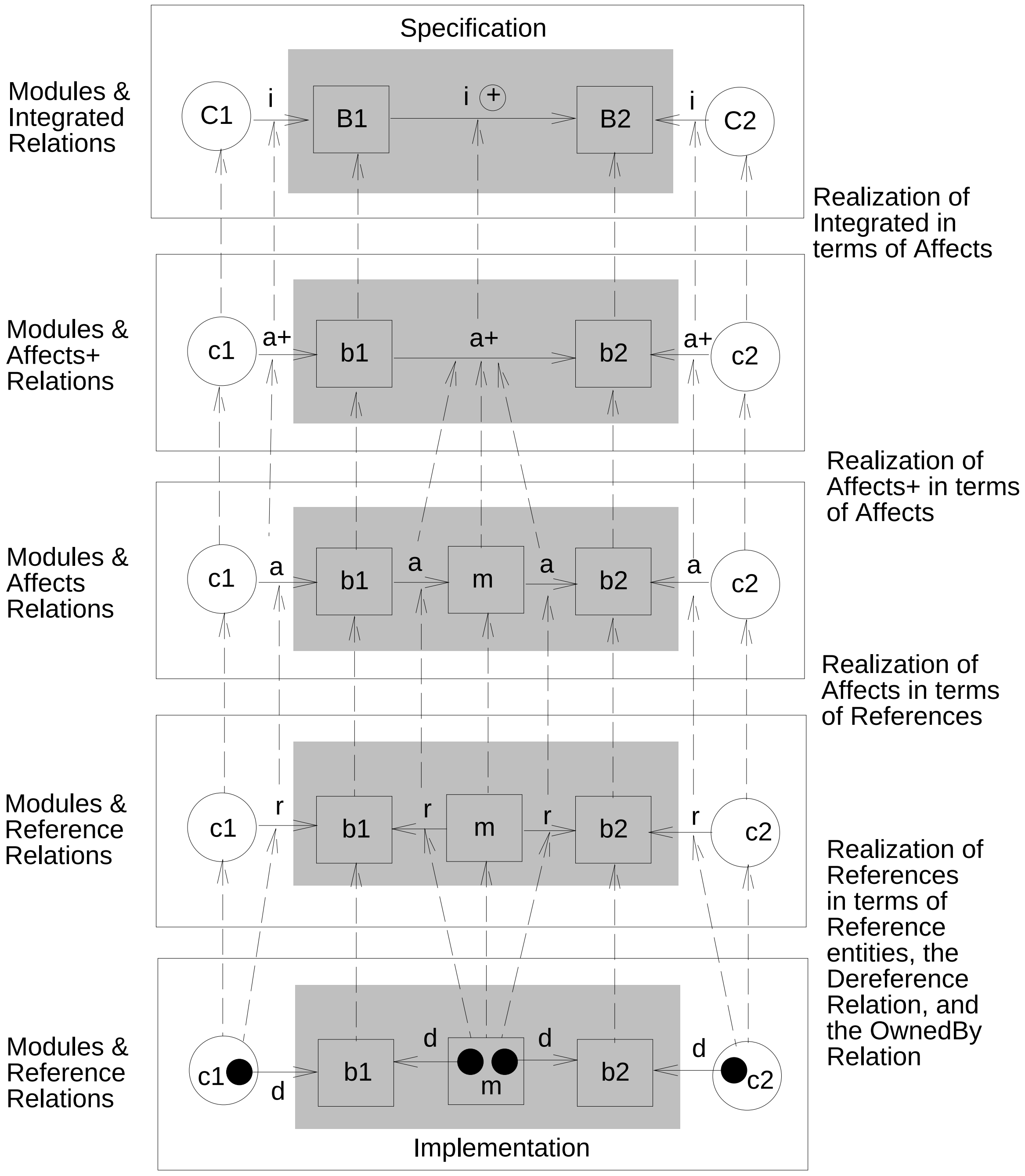
Suppose that we already have objects $b1$ and $b2$ implementing behaviors $B1$ and $B2$. How do we evolve these objects into an implementation of $B1 \oplus B2$. The rest of this section combines the requirements given above with the axioms of the modeling framework to derive a method that produces implementations in which such changes are realized by the addition of mediator objects to implementations, as required.

5.3.1 The Basic "Design Move"

Figure 5.2 presents the "kernel" of the mediator method by diagraming the sequence of design decisions leading from the behavioral ER model $B1 \oplus B2$ to a mediator-based implementation that realizes this integrated the behavior.

We start by using the framework to model $B1$ and $B2$ as modules and $\oplus$ as an element of the *integrated* relation (incident on the modules $B1$ and $B2$). One requirement on the design method we seek is that it relieve clients of the burden of integration. This is done by drawing the *affects*$^+$ relation needed for integration between $b1$ and $b2$. We need a chain of *affects* relations (invocations) from $b1$ to $b2$.

A second requirement (imposed by the definition of transitive closure) is that the *affects*$^+$ relation be realized by a chain of *affects* from $b1$ to $b2$. Combining this with the requirement that the design method relieve the objects to be integrated from the burden of integration leads to the conclusion that the chain must start with an implicit invocation from $b1$ (otherwise a reference to the invoked module would have to be added to $b1$). Similarly, the invocation chain must end with an explicit invocation of $b2$ (since implicit



Specification
Modules & Integrated Relations
C1
i
B1
i ⊕
B2
i
C2
Realization of Integrated in terms of Affects
Modules & Affects+ Relations
c1
a+
b1
a+
b2
a+
c2
Realization of Affects+ in terms of Affects
Modules & Affects Relations
c1
a
b1
a
m
a
b2
a
c2
Realization of Affects in terms of References
Modules & Reference Relations
c1
r
b1
r
m
r
b2
r
c2
Realization of References in terms of Reference entities, the Dereference Relation, and the OwnedBy Relation
Modules & Reference Relations
c1
d
b1
d
m
d
b2
d
c2
Implementation

Figure 5.2: The mediator method analyzed as a set of design decisions.

invocation would require addition to $b2$ of a reference to some announcing module). We are thus driven to add a new module to the implementation with the chain of *affects* passing through it. We add one such module that is implicitly invoked by $b1$ and that explicitly invokes $b2$. It thus references both $b1$ and $b2$. This is the mediator module that realizes the behavioral relationship $\oplus$.

5.3.2 Structural Statics and Dynamics

A key static structural feature of this implementation is that neither switch object has to reference any another module or manage information related to integration. Integration is achieved without compromising the independence or simplicity of the switch objects. Rather, the required references and other code and data are held by the mediator m. See the bottom panel in Figure 5.2. In other words, this design separates and modularizes integration concerns in the implementation in a manner that parallels the separation of the behavioral relationship in the behavioral ER model.

This separation of the relationship in the implementation corresponds to a direct realization mapping between the implementation and the behavioral ER model. Following the chains of arrows representing realization relations upward in Figure 5.2 shows that the mediator m realizes the relationship $\oplus$ as a unit. The composition of the arrows—i.e., the realization mapping—is shown in Figure 5.3. The mediator method produces multi-representational systems having this kind of "continuity" in their static structures.

This mediator method yields implementations whose dynamic structures also parallel those of the corresponding behavioral ER models. In this case, the switch and client objects of the unintegrated system are used directly and without change in the integrated system, and the addition of $\oplus$ to the specification is paralleled by the addition of the mediator m to the implementation.

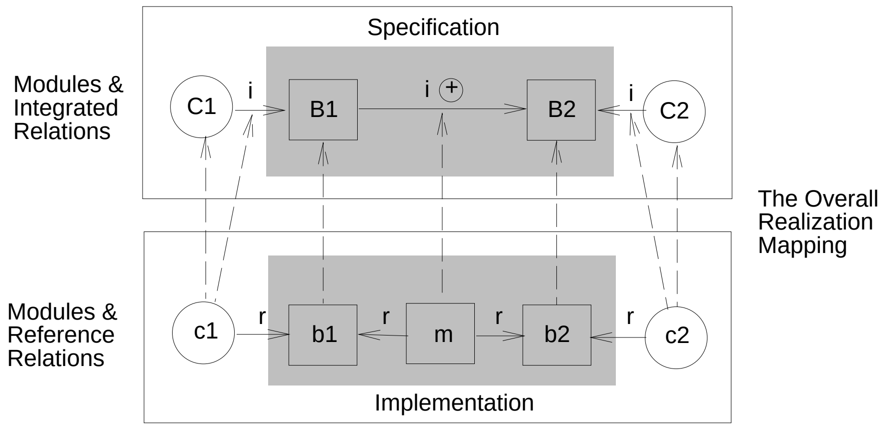

Specification
Modules & Integrated Relations
C1
i
B1
i ⊕
B2
i
C2
The Overall Realization Mapping
Modules & Reference Relations
c1
r
b1
r
m
r
b2
r
c2
Implementation

Figure 5.3: The direct realization mapping between a behavioral ER model and a mediator implementation of the model.

5.3.3 The Prototypical Trigger Mediator

This example illustrates both a prototypical *trigger* behavioral relationship, *when $B1$ is turned on, turn on $B2$;* and a trigger mediator that implements this relationship as a separate, external object. To make the properties of this design concrete, it helps to compare it with designs produced using common methods. Using hardwiring, a client calls $b1$, and $b1$ calls $b2$ to turn it on. In the implicit invocation style, $b2$ references $b1$ to be implicitly invoked by it. In a message server design, $b1$ explicitly invokes the BMS, and $b2$ references the BMS to be implicitly invoked by it. An encapsulation design requires clients to call a wrapper component w that explicitly invokes $b1$ then $b2$.

In the mediator design, by contrast, the mediator is implicitly invoked by $b1$, and the mediator responds by explicitly invoking $b2$. Fundamentally the mediator method *combines* the indirection afforded by a separate mediator object with the careful use of implicit and explicit invocation to preserve the independence of the objects being integrated. There is more to mediators than indirection. The mediator method is the only one of the methods discussed to realize the behavioral relationship as a separate module, to preserve the independence of $b1$ and $b2$, and to relieve their clients of the burden of integration.

5.4 The Abstract Behavioral Type

This section addresses the claim that the ADT combined with the exclusive use of explicit invocation for communication is an unduly restrictive basic building block for constructing integrated systems. (I will refer to this basic building block as *the ADT* for short. This section presents the abstract behavioral type (ABT) as an enriched basic building block that does support the mediator method in designing real-world integrated systems.

5.4.1 Shortcomings of the Abstract Data Type

The problem with the ADT is that it essentially abstracts behaviors solely in terms of operations applicable to type instances; and in practice this seems to lead to explicit procedure call as the primary means of linking ADT-based objects. Explicit invocation is the primary or only supported mechanism for communication among ADT-based objects in many programming languages and systems. In particular, the ADT does not make dynamic behaviors that objects undergo "audible" to external observer objects, even if some of those behaviors are abstractly meaningful; nor does it provide any means for external objects to respond to the dynamic behaviors of other objects.

Given the restriction imposed by a lack of support for implicit extension of behavior, hardwiring and encapsulation methods are sensible approaches to integration. The problems with these methods (discussed above) suggests there is something wrong with or missing from the concept of the ADT. The missing ingredient is implicit extension—the audibility in the external environment of abstractly meaningful behaviors in which objects engage, and the ability for other objects to respond to such behaviors.

Of course, supporting implicit invocation is not enough by itself to ease the design of integrated systems. Common methods that support implicit invocation tend to yield hardwired structures. In the Smalltalk-80 MVC method, for example, view objects explicitly invoke and register with models. The ADT idea haunts integrated system designers, discouraging use alternate architectural styles even when they are at hand.

5.4.2 Making Events Dual to Operations

Despite its shortcomings, the ADT provides a rich framework for conceiving and representing complex behaviors. We we would be loath to give up its advantages. I thus present what I call the abstract behavioral type (ABT) as a better basic building block: one that repairs the shortcomings of the ADT rather than abandoning the ADT in favor of a more radical alternative.

The ABT extends the ADT by placing events (audible behaviors) at the same level of abstraction as operations. ADT operations are first-class; and so in the ABT, events are also first-class. An ABT interface defines both operations and events. Since ADT operations have names, signatures, and semantics, so do ABT events. Thus, an ABT characterizes a class of objects in terms of both the applicable operations and the audible events; and in addition to explicitly invoking each operations, ABTs permit operations to be implicitly invoked by other objects' event announcements.

In contrast to the ADT, the ABT encourages the designer to think and to express thoughts in terms of objects characterized not only by applicable operations but also by audible (but abstract) behaviors. Thus, a switch object can not only be turned on and off, but other objects can observe and respond to a switch's going on and off. The ABT raises implicit invocation as found in FIELD and Smalltalk-80 from the status of low-level imperative communication mechanism to that of high-level abstraction mechanism.

While this extended basic building block provides the designer with a more flexible material from which to compose complex systems, by itself, it does not overcome the problems with common design approaches. It provides a rich abstraction mechanism for implementations, but it does not encourage the designer to analyze complex behaviors in terms of simpler behaviors and behavioral relationships. The full benefits of ABTs are realized when one thinks about integrated systems in terms of behavioral ER models, and when one then uses ABTs to realize behavioral ER models in implementations with corresponding static and dynamic structures. This combination of ideas is at the heart of this work.

5.5 Implementing Behavioral ER Models Using ABTs

The key property of the ABT, in contrast to the ADT (with respect to this work) is that the ABT supports conception and expression of both behaviors and behavioral relationships as objects. Thus, behavioral ER models can be implemented directly, with one ABT instance realizing each behavior and one ABT instance realizing each behavioral relationship in the model. Not only does the static structure of the implementation follow that of the model, but the dynamic structure does, as well. As behaviors and relationships are added to, deleted from, and changed in the behavioral ER model, the corresponding addition, deletion and modification of ABTs suffices to keep the implementation consistent. This section illustrates this key idea.

5.5.1 Designing and Representing Behaviors as ABTs

Figure 5.4 defines a *Switch* ABT in a notation resembling C++ [Stroustrup 86]. An instance of this type has the expected operations; but it also has two events whose announcements signal transitions between the on and off states (caused by the execution of the *TurnOn*() and *TurnOff*() operations). Both the operations and events are given names, parameters, and return values. The semantics are simple. The *TurnOn*() operation, applicable when the switch is off, turns the switch on and "announces" the *WentOn*() event. *WentOn*() is announced if and only if the switch state toggles from off to on. The *TurnOff*() operation and *WentOff* event are symmetrical. This behavior is implemented in an imperative style by implementing the *TurnOn*() operation as in Figure 5.4 (similarly for *TurnOff*()).

5.5.2 Designing and Representing Behavioral Relationships as ABTs

Having conceived of and implemented the switch behaviors as ABTs with the three operations and two audible behaviors, we now devise a completely separate mediator ABT to implement the behavioral relationship $\oplus$. See Figure 5.5. This ABT is called *Oplus*. The *Initialize*() operation of the ABT places the given switch objects into the

```
class Switch {
  state:
    bool SwitchState;
  operations:
    void TurnOn()  { State = ON;  Announce WentOn(); }
    void TurnOff() { State = OFF;  Announce WentOff(); }
    bool IsOn()    { return SwitchState; }
  events:
    void WentOn();
    void WentOff()
}
```

Figure 5.4: Outline of an abstract behavioral type for the switch behavior. This code declares a class of switch objects having a state, either on or off; operations to toggle and observe the state; and events indicating when the state goes on or off. The implementation of *TurnOn* is presented. It changes the switch state then announces the *WentOn* event. *Announce* is a keyword for announcing the designated event (thereby implicitly invoking operations registered with the event).

relationship by setting variables $S1$ and $S2$ to reference the objects, and by registering *UponS*1*WentOn*() to be invoked implicitly by the announcement of the *WentOn*() event of $b1$ (referenced by $S1$). Thereafter, announcement of this event implicitly invokes *UponS*1*WentOn*(), which implements the relationship by calling $b2$ (referenced by $S2$) to turn $b2$ on.

This method integrates the behaviors of the objects without imposing on either the objects themselves or their clients. Consider the program fragment in Figure 5.6. The behaviors of $b1$ and $b2$ are integrated by the mediator *op*. When any client turns on $b1$ (in this case the main program itself) the mediator turns on $b2$.

In comparison to a hardwired design, this example moves the invocation of $b2$ by $b1$ out of $b1$ and into a mediator, thereby "externalizing" the explicit invocation. Despite th simplicity of this example, its strength is that it provides a template, a structuring paradigm, that can be used to organize representations of much more complex behaviors.

```
CLASS Oplus {
  STATE:
    Switch* S1;
    Switch* S2;
  OPERATIONS:
    void UponS1WentOn() {S2->TurnOn(); }
    void Initialize(Switch* b1, Switch* b2) {
        S1 = b1; S2 = b2;
        REGISTER(UponS1WentOn,S1->WentOn);
    }
}
```

Figure 5.5: A mediator implementing the behavioral relationship $\oplus$.

```
void main() {
  Switch b1;
  Switch b2;
  Oplus op(b1,b2);
  b1->TurnOn();
}
```

Figure 5.6: A mediator-based implementation of the system $B1 \oplus B2$.
This C++-like program implements the integrated behavior $B1 \oplus B2$ by integrating representations of the individual behaviors. Specifically, the presence of *op*, which represents the behavioral relationship $\oplus$, is sufficient to integration the independent switches $b1$ and $b2$.

```
void main() {
  Switch b1;
  Switch b2;
  Oplus op(b1,b2);
  b1->TurnOn();

  Switch b3;
  Otimes ot(b3,b1);
  b3->TurnOff();
}
```

Figure 5.7: A mediator-based implementation of $B3 \otimes B1 \oplus B2$.
This C++-like program is a follow-on version to the implementation of $B1 \oplus B2$. This representation differs precisely by the addition of new "modules" corresponding to the new behavior and behavioral relationship.

5.5.3 Organizing the Evolution of Integrated Systems

In addition to organizing the static structure of the integrated system implementation, the method also organizes its dynamic structure—the way the structure changes through execution and evolution. Consider the step taking the system $B1 \oplus B2$ to $B3 \otimes B1 \oplus B2$. (Recall that $\otimes$ requires switches $B1$ and $B3$ to operate in tandem.) The behavioral ER model evolves by addition of a vertex for $B3$ and an edge for $\otimes$. The implementation evolves by the corresponding addition of a third switch object and a second mediator object. The third switch is a new instance of the existing *Switch* ABT. The mediator is an instance of a new mediator ABT *Otimes*. See Figure 5.7.

Figure 5.8 outlines the *Otimes* mediator. Like *Oplus*, this mediator references the objects whose behaviors it integrates. It has an additional bit, *busy*, that it uses to avoid circular updates, as explained below. *Otimes* defines four operations (whose names all start with *Upon*) that it registers to be implicitly invoked by the events of the switch objects. The mediator defines an initialization operation that makes the switch objects consistent when they enter the relationship. It does this by changing the state of the $S3$ switch, if necessary, to ensure that the required invariant holds to start. The initializa-

```
class Otimes {
  State:
    Switch* S1;
    Switch* S3;
    bool busy;
  Operations:
    void UponS1WentOn() {
      if (busy == FALSE)
      {
        busy = TRUE;
        S3->TurnOn();
        busy = FALSE;
      }
    }
    void UponS1TurnOff();
    void UponS3TurnOn();
    void UponS3TurnOff();
    void Initialize(Switch* b1, Switch* b3) {
      S1 = b1; S3 = b3;
      if (S1->IsOn() != S3->IsOn()) {
        if (S3->IsOn())
          S3->TurnOff();
        else
          S3->TurnOn();
      }
      Register(UponS1WentOn,S1->WentOn);
      Register(UponS1WentOff,S1->WentOff);
      Register(UponS3WentOn,S3->WentOn);
      Register(UponS3WentOff,S3->WentOff);
    }
}
```

Figure 5.8: A mediator implementing the behavioral relationship $\oplus$.

The Initialize operation places the given switch ABT instances into the desired behavioral relationship. The state variables $S1$ and $S3$ are made to reference the given switches, and $UponS1\,WentOn()$ to be registered to be implicitly invoked by $S1's\ WentOn()$ event. Thereafter, announcement of $S1- > WentOn()$ implicitly invokes $UponS1\,WentOn()$, which turns on $S3$.

tion operation also performs the registrations and other such activities to establish the relationship. Once this is done, the mediator then automatically manages the behavioral relationship.

To see how this implementation works, suppose that $b1$ is off and that is gets turned it on. This causes $b1$ to announce its *WentOn* event. Since both mediators have operations registered with that event, each is invoked, in some unspecified sequential order. (Event announcements are broadcast.) The $\oplus$ mediator turns on $b2$, while the $\otimes$ mediator turns on $b3$. The latter action then causes $b3$ to announce its *WentOn* event, which re-invokes the $\otimes$ mediator. However, the mediator operations are guarded in a way that prevents circular actions from being taken. The first invocation of the *Otimes* mediator sets the *busy* flag, so when the second event is received from $b3$, the event is dismissed. Control returns to $b3$, then back to *Otimes*, and finally to $b1$, from which the initiating operation returns with the whole system in a globally consistent state.

Now consider further evolution: the change of $\otimes$ to $\otimes'$. (Recall that when in eager mode $\otimes'$ requires $B1$ and $B3$ to act in tandem.) The behavioral ER model changes by modification of the $\otimes$ edge. To make the implementation consistent requires changes only to the corresponding *Otimes* mediator module. We add an eager/lazy component to the mediator's state; we add operations to its interface to permit clients to toggle this state; and we add code to the operations implicitly invoked by $B1$ and $B3$ to consider this state. When it is lazy, the operations just return without taking any action.

5.6 Conclusion

The mediator-based implementation of the behavioral ER model for $B3 \otimes B1 \oplus B2$ provides an archetypal example of the mediator method. It illustrates how both triggers and state invariants can be thought about and implemented in this manner. The following two chapters show the extent to which the simple ideas presented in this chapter can be profitably used in the design of much more complex systems.

Chapter 6

Validation

Validating claims for software design methods is difficult. This work presents a new method and claims that it is better than common methods. How can such a claim be validated? Not only is hard to compare design methods on intrinsic, technical grounds; but the practical value of any new method depends on the context of practice into which it is introduced. The state of the people who might use the method is a key factor in the success of the method—independently of its intrinsic, technical merits. Thus, in addition to demonstrating technical merits, one should show that a proposed method can actually be introduced into situation of practice and then be used in a consistently profitable way.

I therefore take a two-pronged approach to validating the claim that the mediator design method is better than common design methods with respect to ease of design, realization, and evolution of integrated systems. The first prong is rational. It validates the mediator method by arguing for its intrinsic, technical properties, as revealed by intellectual reasoning from simplifying models and examples. The second prong is experiential. It validates the mediator method through thoughtful reflection on a range of experiences in which the method was used in practice—by myself and others. Although these experiences do not "prove" anything, collectively they provide convincing evidence for both the novelty and value added by the mediator method.

6.1 Rational

The whole preceding part of this dissertation presents the intellectual arguments for the claims that common design methods complicate integration and evolution and that the mediator design method overcomes this problem. The foundations on which the arguments are based—namely the evolutionary scenario, the structure modeling framework, and the characterizations of design methods and their evolutionary behaviors—have been defined and presented rigorously enough to be carefully checked. Do predictions about common design methods match experience? Do the framework and evolutionary scenario model reality well enough to be useful for the intended purposes?

Two primary accomplishments help to validate the modeling framework and the evolutionary scenario as modeling constructs. First, they enabled useful characterization of common methods and the demonstration that these methods unnecessarily complicate integration and evolution. Second, the framework and scenario supported derivation of a new design method that overcomes the problems with common methods.

The preceding chapters have shown that common software design methods unnecessarily and significantly complicate integration and evolution. Common methods impose unnecessarily large lower bounds on the complexity of the static and dynamic structures of integrated systems—from the simplest ones, as exemplified by the evolutionary scenario, on up. The preceding chapters have also defined a new design method intended to avoid these complications; and they framed a hypothesis about that method: that it provides significant help in practice. The preceding chapters, however, have not presented evidence based on real world experience to substantiate this claim. That is the task of the following section.

6.2 Empirical

Relying on the intellectual arguments alone as a basis for investing in new software design methods is dangerous, and—given the potential costs and liabilities involved—

even foolish. The complexities of developing real integrated systems are great enough that it is easy for abstract theories to overlook or underestimate technical and human matters that are critical in practice.

As in any design discipline it is critical to show that a new method really works in its intended context of practice. Does the mediator method really ease the lower bound on complexity when applied to real integrated systems? Does it really help people to be better designers by giving them a way to think more clearly about the behaviors and implementations of integrated systems? The answers to these question require that the abstract ideas be tested in practical application to the design of real integrated systems. The proof is in the pudding, as the saying goes.

The "pudding," to date, comprises a set of integrated systems designed and implemented using the mediator method. Some of these were built by me; others by colleagues. The systems include a prototype parallel programming environment, a prototype computer-aided geometric design environment; a semantics-preserving program restructuring tool, a production-quality computer-aided radiation treatment planning system, a prototype computer-aided design system running under a single large address-space operating system, and a commercial, multi-scale geographical information browser.

The following subsections provide brief overviews of these systems. For each one, I discuss the ways in which the experiences of building the systems helped validate the claims made for, and to illuminate aspects of, the mediator approach. The next chapter presents some of these arguments in greater depth, in the context of a detailed case study of the mediator method, as it was used in designing the Prism radiation treatment planning system.

6.2.1 Parallel Programming Environment

The first system in which I applied the mediator method was a prototype environment for programming large-scale, nonshared-memory parallel computers [Snyder 89]. At the highest level, a program in this environment is structured as a sequential composition of *phases.* Each phase is structured as a *graph* that represents a parallel program to be run on a network of computing nodes. Each vertex in the graph specifies a sequential *program* to run on a single processor. Edges represent communication channels. Concretely, these three-tiered programs are represented as collections of objects in the C++ programming language.

The environment supports four interactive tools for manipulating programs. The phase and graph tools are graphical depict programs as flow-charts and interconnection graphs, and allow the user to build and modify programs in a direct manipulation, graphical style. The process tool, which is used to write the sequential programs, is text-based. The fourth tool provides a top-level user interface that allows instances of the other tools to be created and deleted dynamically.

The phase, graph, and process tools are related to their respective parts of a program (represented as a set of objects) by mediators that keep the depictions and the program consistent. Furthermore, the three levels of the program are related to each other by mediators. If the name of a sequential process is changed, all graphs having processors that execute that process are updated accordingly. Those updates are then propagated to the interfaces of all active graph tools by other mediators.

This structure allowed separate designers independently to develop the parts of the data base and the tools that operate on them. Once the tools and their respective parts of the data bases had been developed and tested, we integrated them by adding mediators to keep related representations consistent.

6.2.2 Computer-Aided Geometric Design System

Tom McCabe developed a prototype "kernel" for a computer-aided geometric design (CAGD) system [McCabe 91], supporting interactive, multi-view editing of a *mesh.* A mesh is a discrete representation of a surface—technically a two-manifold—embedded in an n-dimensional Euclidean space. A mesh is similar to a graph, but it has topology: edges are ordered around vertices. A mesh M is represented as a vertex set VS and an edge set ES plus an object M that integrates VS and ES. Vertices in VS have property lists that store information such as vertex geometry and labels. ES stores a topological algebra—a set of edges plus several ordering relations over this set. Specifically, ES implements a variant of the *quad-edge* data structure [Guibas and Stolfi 85]. The clean, somewhat complex algebraic structure of this representations made it useful for exploring the consistency-maintenance capabilities of mediators.

McCabe's system supports viewing of a mesh with n-dimensional geometry ($n > 2$) as a set of 2-D views presenting orthographic projections. Views are integrated with the mesh in relationships that ensures that as a mesh changes, views are updated, and that as elements are moved in views, the mesh is updated. Although views are not inherently tied to meshes, in this system, as a vertex is moved, the views animate the motion from their own perspectives, owing to the presence of various mediators of several kinds. One kind of mediator projects n-dimensional meshes into two dimensional meshes. Another integrates 2-D meshes with views based on the InterViews toolkit [Linton, Vlissides, and Calder 89]. A third kind of mediator keeps "dots" on the screen consistent with mesh vertices as either changes.

This system has several structural properties that help to validate the claims made for the mediator method. One property is that topological and geometrical information is separated in both conception and implementation. This decomposition simplifies the task of designing more tools. A tool to manage topology would operate on ES alone. Another could specify geometry, e.g., layout, operating on VS alone. A third could compute smooth surface representations (requiring topology and geometry) using M.

Another useful structure is the isolation of the relationships between the 2-D vertices that are displayed and the corresponding n-D vertices in meshes. Because mapping from n to two dimensions discards information, the reverse mapping from two to n is ambiguous. The question is how to change the n-D vertices when the 2-D vertices are changed. McCabe solved this by defining the mediators to "fill in" the missing information. The mediators map 2-D movements to n-D movements that are parallel in orientation to the

2-D viewing plane. Mediators provide a place to encapsulate designer-defined policies for resolving ambiguity in such “back mappings.” This approach increases flexibility over systems that constrain designers by fixing a policy for handling ambiguity.

6.2.3 Semantics-Preserving Program Restructuring Tool

Griswold used the mediator method to design in several systems. The first was a tool to ease software enhancement by supporting meaning-preserving restructuring of programs [Griswold 91]. The tool helps the software engineer by decomposing enhancement into two phases. The claim is that prior restructuring decreases the cost of subsequent semantic enhancement more than enough to offset the up-front restructuring cost.

One transformation changes the order of formal parameters in a procedure declaration. When the tool applies this operation, the tool preserves meaning by swapping the actual parameters at all call sites, unless that change would not restore the meaning—e.g., if evaluating the parameters has side effects whose order would be changed. In the latter case, the tool aborts the operation, leaving the program unchanged.

The heart of this system is an integrated pair of program representations. An abstract syntax tree (AST) represents the syntactic program structures. A program dependence graph (PDG) represents static semantic information. The representations are integrated by a mediator that keeps them consistent and that makes the relations between them available for use by the higher levels of the tool. Thus, the tool maps from a procedure declaration node in the AST to a corresponding PDG node, from which call sites can be found by traversing the PDG.

One point relevant from the perspective of this dissertation is that the mediator method helped Griswold find his design—with the semantically complex relationships between representations external to both of them and to the higher-level aspects of the tool. The mediator method also allowed Griswold to reuse a substantial, existing software component: over 10,000 lines of Common Lisp/CLOS [Bobrow et al. 88] comprising the PDG, which was built by Jim Larus as part of his Curare system [Larus 89].

Localizing the relationship between the AST and the PDG in the mediator was also useful in providing a place for the code and data that implemented a somewhat complex approach to propagating changes from the AST to the PDG. The mediator caches notifications from the AST and computes and propagates changes to the PDG only upon demand from the higher levels of the tool. This illustrates the implementation of *slippage* in maintaining the consistency of representations (as in the relationship $\oplus'$ described in Chapter 2). Because the mediator is a first-class object, it is easy to have it maintain an "inconsistency buffer." While the AST and PDG are inconsistent at some points in time, the system as a whole is consistent in that mediator has stored and can thus resolve the AST/PDG inconsistencies.

6.2.4 Radiation Treatment Planning System

In the efforts discussed above, either I or close colleagues controlled the requirements for the systems to be built, leaving some doubt as to whether the requirements for which the mediator method is appropriate are limited to those carefully crafted for a mediator implementation. My experiences with Prism and with the geographical information system discussed below allay this doubt. Prism in particular [Sullivan, Kalet and Notkin 93] is a thoroughly documented, substantial example of a system for which requirements were fixed prior to initiation of contact with the mediator method.

In this case, Ira Kalet—a researcher in the University of Washington Radiation Oncology Department and an accomplished designer of radiation treatment planning systems [Kalet and Jacky 82, Jacky and Kalet 86, Jacky and Kalet 87b]—had written requirements for a third-generation, integrated treatment planning system. This system was to be more flexible and tightly integrated than any such system previously built.

Kalet saw that common design methods would lead to an artifact far too complex and costly to be built on the limited budged available. So, when he heard about the mediator method, he decided to try it. I began to work with Kalet about once a week over a period of about a year analyzing the requirements by crafting behavioral ER models, designing infrastructure (e.g., graphical user interface widgets, an implicit invocation mechanism, and basic ABTs and mediators [Kalet 92]), and advising on details of implementation.

My contribution was the architecture of Prism. I was not always involved in detailed design or implementation. So, in addition to providing a fair test of the mediator method, Prism also provided experience transferring the mediator method to another experienced designer and his colleagues. Both the system and transfer were successful. The next chapter presents Prism as an in-depth case study, so I will not discuss it further here.

6.2.5 Large Address-Space Operating Systems

Chase *et al.* exploited the mediator approach in the context of a single, large, shared address-space operating system, Opal [Chase *et al.* 94]. In this system, address translation and memory protection are decoupled. One promise of this kind of system is to efficiently support computer-aided design of such complex artifacts as passenger jets.

Chase *et al.* observed that by tightly coupling address spaces and memory protection, common operating systems impose unnecessarily costly tradeoffs among protection, integration, efficiency. Protecting tools running them in separate processes increases the cost of integration (in both code complexity and runtime efficiency) by requiring cross-address space invocations, for example.

The arrival of inexpensive 64-bit microprocessors opens opportunities for operating system structures that ease this problem. One key advance is that 64-bit virtual address spaces are large enough to accommodate many tools operating on large data sets–for example, CAD systems working on complex artifacts such as passenger jets.

The idea is to run all the CAD tools to be integrated in the same address space but in different, and possible overlapping, protection domains. Being in the same address space makes it possible for tools to reference each others' data directly, which lowers program complexity and possibly runtime overhead. A tool running in one domain that has read access to another domain can get data from tools in that domain without requiring expensive protection checking.

To demonstrate the potential for this approach, Chase *et al.* built a prototype environment with two tools are integrated by a mediator. When changes are made to a "CAD database," the mediator, which runs in its own protection domain, is invoked. The mediator, in turn, updates a "derived database" tool running in a third protection domain. To avoid the need for the CAD tool to know about the mediator, the implicit invocation system underlying this prototype gives the protection domains in which the "CAD database" tool runs write access to the mediator, allowing the tool to implicitly invoke the mediator using a local procedure call. The mediator has read access to the CAD tool. It can fetch data needed to recompute the "derived" data, again using a local procedure call. The mediator cannot corrupt the CAD tool, however, because it does not have write permission. Nor does the mediator have access to the domain in which the derived tool runs, so it uses an expensive protected procedure call to communicate with it. However, this solution is better than any of those available on more traditional operating platforms.

How does the experience of Chase *et al.* help validate the work in this thesis? There are two basic answers. First, it documents a case in which designers other than the creators of the mediator method successfully used the method to decompose an integrated system into a system in which two separate tools are tightly integrated. Second, it provides an example of how the approach can be employed in operating systems contexts different from the rest of the experiences reported here—where all "tools" run in a single, Unix process. Further efforts in this direction would be useful.

6.2.6 Multi-Scale Geographic Information Browser

C. Brett Bennett, an experienced commercial software developer in Seattle, used the mediator method to architect a multi-scale geographic data browser for the real estate industry.[1] The system supports several tools for exploring the properties for sale in large, metropolitan areas. The system presents the user with an overview street map,

[1]Personal communication with C. Brett Bennett.

and allows zooming and panning to explore the displayed region. At different scales, different kinds of information are presented. Only large roads appear in wide angle views, for example, while small streets and property icons appear when the user zooms in. As the mouse passes over icons, corresponding street addresses, other textual information, and images are displayed.

Bennett claims that the mediator method led to a significantly better architecture than he would have obtained without it. He values the decomposition of his system into simple, independent, reusable parts, and the highly flexible structure of the resulting program. He also believes he has devised a set of generic behavioral and mediator objects that will enable him to efficiently assemble a range of follow-on products.

On the cost side, Bennett claims that the mediator method requires greater care in defining the semantics of components—especially the creation and deletion of mediators at runtime. He found that lack of care at the design stage led to difficult debugging problems laterd—problems harder to fix than those that arise in traditional designs.

6.3 Conclusion

Each of these development efforts was successful. Behavioral ER modeling was used to design mediator-based implementations that met the requirements given for tightly integrated behaviors. Moreover, each of the systems had the desired, beneficial static and dynamic structural properties—a clean decomposition and a flexible structure.

In terms of static structure, each system comprises separate behavioral objects, used directly by clients, and integrated by separate mediator objects. In execution dynamics, dynamically generated behavioral objects are integrated by dynamically instantiating mediators. Finally, although the data is sparse, the systems built this way appear to have the expected evolution dynamics as well. For example, at least one major tool that was not originally specified for the Prism system, was built and integrated in the expected way. The tool was built as an independent entity. Mediators were then defined to integrate its behavior with the behaviors of the rest of Prism.

Despite the anecdotal character of these reports—the lack of precise hypotheses, precisely defined measurements, experimental designs, carefully collected quantitative data, or statistical hypothesis testing—despite these shortcomings, the reported experiences with the mediator method lend significant credibility to the claim that this method does ease development of serious (albeit not immensely complex) integrated systems. Moreover, the design, realization, and evolution tasks are eased in the ways predicted by the reasoning presented above. The mediator method is no "silver bullet [Brooks 86]," but it provides significant value by helping software designers to think more clearly about integrated behaviors and to avoid unnecessary complexity in the design and evolution of software systems that realize those behaviors.

Chapter 7

Prism: A Case Study

This chapter presents a case study describing how behavioral ER modeling and mediators were used in designing and implementing Prism, an integrated system for planning radiation treatments for cancer patients in clinical use at the University of Washington Cancer Center [Sullivan, Kalet and Notkin 93, Kalet et al. 91, Kalet et al. 92]. The purpose of this chapter is to substantiate the claim that the mediator method can substantially ease the design and realization of integrated systems. Prism has not been in use for long enough to provide a basis for strong claims about ease of evolution.

7.1 Introduction

Prism was developed in an interdisciplinary collaboration between software engineering researchers (David Notkin and myself) and radiation oncology researchers (Ira Kalet and his colleagues). Each group had its own goals. Kalet had specified requirements for a radiation treatment planning (RTP) system that would be richer and more tightly integrated than any previously built; and he needed to realize this system on a modest budget. Sullivan (with Notkin) sought experience to help validate the claim that behavioral ER modeling and mediators do ease the design, realization, and evolution of integrated systems.

Both goals were substantially met (although, as noted above, data on the evolution of the system is still sparse). The mediator method provided Kalet with leverage he needed to effectively design and implement a system more ambitious than could have been affordably built using common design methods. His success supports the claims of this work. Prism implements Kalet's original specification, with minor changes made during detailed design. The tools support diverse planning tasks, including modeling of patient cases (anatomy, images, treatment plans), computation of target volumes, calculation of radiation dose distributions, visualization, and database management. The tools work together in fine-grained, dynamic interactions. Kalet attributes the success of the effort to the conceptual and structural benefits of behavioral ER modeling and mediator design.

To give a sense of what a dosimetrist might sees when using Prism, Figure 7.1 presents a screen dump. This view of the workstation during system use pictures a set of panels displayed at one point during a session. Several visualization panels display the anatomy of a patient in various cross sections, with two radiation beams passing through (the diverging pairs of lines). A tool panel for changing the parameters of one of the beams is also displayed (the panel with the dials). When the user moves the dials, the beam parameters are changed. When this happens, the graphical views are updated. This figure is discussed in more detail below. The next section addresses background issues: the radiation treatment planning domain, and the state of the art before Prism.

7.2 Radiation Treatment Planning

An RTP system is a set of software tools used by a dosimetrist to design radiation treatments for cancer patients. Dosimetrists are experts in designing radiation doses. A treatment is intended to deliver a prescribed dose of radiation to target regions containing cancerous tissues without overdosing surrounding tissues. A treatment plan basically defines a three-dimensional configuration of radiation beams (and other sources) in relation to a patient's anatomy crafted to deliver the prescribed dose.

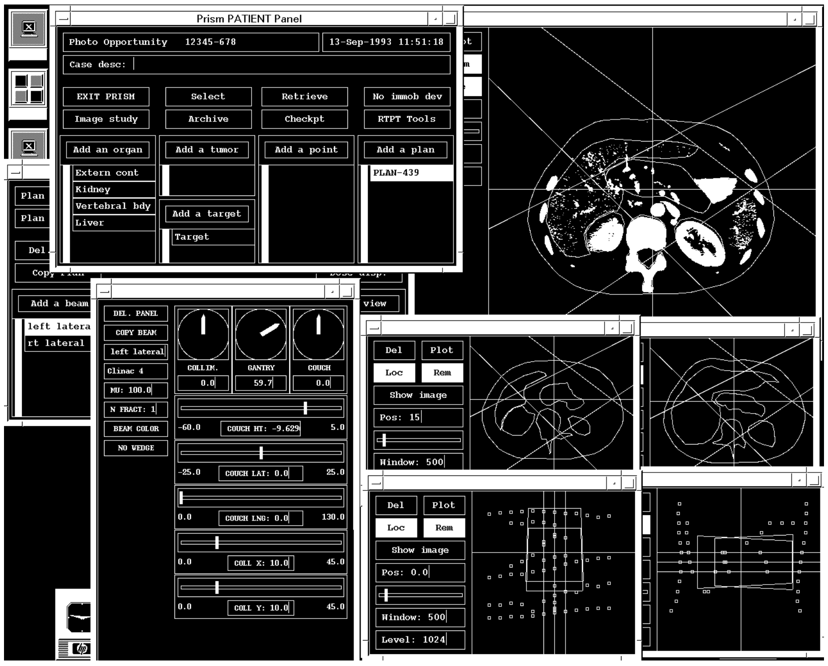


Photo Opportunity 12345-678

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 7.1: A Prism screen.

Designing effective treatments can be hard. The space of treatment plan configurations is huge. Different people are differently shaped. People move during and between treatment phases. Tumor types and locations vary. Treatment machinery differs among installations. Challenging cases require careful orientation of radiation sources relative to patient anatomy, visualization of the resulting configurations, computation of the radiation fields generated, and adjustment of plans based on this feedback. Designing plans manually or using poorly integrated tools is unnecessarily laborious and costly.

By providing computerized tools that automate treatment design tasks, a RTP system not only increases the efficiency with which treatment plans can be designed, but can actually lead dosimetrists to find treatment plans that they would otherwise miss [Rosenman et al. 89]. Tools in such an environment may support management of radiographic images; modeling of equipment, patients, treatments plans; computation of the radiation dosages delivered by treatments; visualization of anatomies, plans, machines, and dosages; management of patient and hospital data; and so on.

The goal of Prism is not only to provide tools to support these tasks individually, but to provide an environment in which all the tools work together to give the dosimetrist a powerful, exploratory planning device. Different views should work together to indicate and permit manipulation of their relative spatial orientations. When models change, computed views should be updated and dose distributions invalidated. The system must be broad in the scope of tasks it supports, tightly integrated, and flexibly configurable—both at runtime and over the longer, evolutionary term.

7.3 Prior Art

Many RTP systems have been built, including several at the University of Washington [Kalet and Jacky 82, Goitein et al. 83, Jacky and Kalet 86, Jacky and Kalet 87b, Fraass et al. 87, Kutcher 88, Rosenman et al. 89]. The problem is that these efforts did not achieve the critical combination of breadth of scope, tight integration, and flexibility. This is not surprising in light of the observation 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].

Taylor's observation has clearly been borne out in the radiation treatment planning domain. First, in many systems, different tasks are handled by stand-alone, "Unix-like" tools that run as separate processes and are loosely integrated through shared files. Kalet's first system [Kalet and Jacky 82] integrated anatomy modeling, beam specification, dose computation, and visualization tools this way. To modify an anatomical model, the dosimetrist terminates the dose display, runs then terminates the anatomy modeling tool, runs the dose computation program, then restarts the dose display.

Second, many systems provide rigid user interfaces. In Kalet's second system [Jacky and Kalet 86, Jacky and Kalet 87b], the user traverses a broad, deep, fixed menu tree to change plans, update models, tailor visualizations, use the patient database, etc. The visualization tools are inflexible, too. The system displays a fixed set of graphical views in a fixed layout on the screen. These limitations make it hard quickly simulate plans.

Third, to the extent that existing systems are tightly integrated, their architectures are often so inflexible as to severely complicate evolution. For example, despite the object-oriented architecture of Kalet's second system [Jacky and Kalet 86, Jacky and Kalet 87b], tight integration of prototype AI-based planning tools [Kalet 92c, Paluszynski 89a] is prohibitively expensive. This architectural inflexibility makes it hard to exploit the results of research by making it hard to integrate new tools, and ultimately inhibits ¡research on software support for radiation treatment planning.

7.4 Requirements for a New Environment

In this context, Prism serves two purposes. First, it supports treatment planning in clinical practice. Second, it provides a platform for research on uses of software technologies in treatment planning.

In the second role, Prism is something like a framework [Johnson 92]. It provides a basic environment can be extended with additional tools. A tool that computes radiation target volumes based on mathematical models of tumor shape and type and patient movement—a tool not foreseen in the original specification—was recently integrated. Integrating this tool was easy, owing to the use of a mediator-based architecture. The tool examines a patient model and computes and updates a target volume object directly. All other Prism tools respond as appropriate.

Both roles—production environment and research platform—demand a combination of tight integration and architectural flexibility. Tight integration is needed for efficient treatment planning. A flexible architecture is also important to support planning. The dosimetrist should be able to invoke tools and configure the environment at runtime, as needed to efficiently plan treatments. The system must support instantiation and integration of tools during a planning session. Architectural flexibility is also needed to accommodate the integration of new tools over the system lifetime—as research concepts are tested and then adopted or rejected.

The strength of Prism is that is combines architectural flexibility—in both execution and evolution—with tight integration during execution. The simple but powerful anatomy modeling tools can stay on the screen with dose display tools. When the anatomy is updated, the dose distribution tool is computes and display the new dose distribution. To help users understand three-dimensional treatment plans, Prism allows any number of views of a given plan; any given view can either be displayed or not; and each view display hints to help the user correlate its relative position with those of other views. 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]

7.4.1 Image Studies

Patient models in Prism are derived from image studies. An image study is a sequences of two-dimensional radiographic images—usually computed tomography or magnetic resonance. Figure 7.2 presents one slice in an image study. We build three-dimensional models of anatomy and tumors by tracing contours of organs, tumors, and targets over adjacent slices. All contours tagged as belonging to a given entity define the geometric model of that entity. These models, augmented with information such as density, cell type, and radiation tolerance, are the basis for planning in Prism.

7.4.2 Master Control: The Patient Panel

Prism tools are presented in *panels.* The *patient panel* is the "master control" for Prism. It is presented to the user when Prism is started. See the upper left of Figure 7.1. This tool is used to select a patient case from the Prism database or to open new one. Once a

[1]For information on obtaining Prism, contact Kalet by electronic mail (ira@radonc.washington.edu).

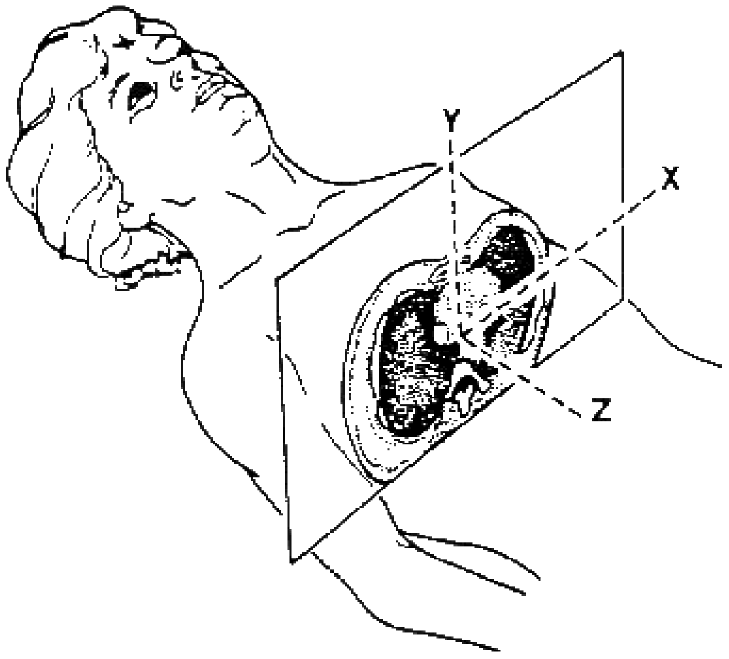


Figure 7.2: A transverse slice.
This figure illustrates the Cartesian patient coordinate system, and shows a transverse slice through the patient. Images in image studies are oriented in this way.

case is opened, associated images can be loaded; administrative data can be edited (e.g., name, hospital ID number); and modeling and planning tasks can be undertaken.

Modeling and other planning tasks are supported in part by the menus (the multiple selection lists that Kalet calls them *selectors*) at the bottom of the panel. Each selector elements pertaining to one aspect of the patient case: organs, tumors, targets, points (which are labels affixed to anatomical points of interest), and treatment plans. Selectors allow these elements to be added, deleted, and selected for editing. Pushing the add button above a selector adds a new element to the associated collection in the patient case model. Selecting an item with the middle button deletes the element. Selecting an element with the left button invokes a tool to display and edit the selected object.

7.4.3 Tool Invocation: The Plan Panel

Selecting a plan, for example, instantiates and displays a *plan panel* to display and edit the designated plan, defined as part of the patient case. The highlighting of "PLAN-439" in the selector above the easel in Figure 7.3 indicates that the plan has been selected and that there is a plan panel for editing it active on the display. The panel is in the lower left of the figure (mostly obscured below the patient panel and beam panel with the three dials). All Prism tools are invoked this way. This point-and-click invocation provides users with the ability to flexibly and dynamically tailor their planning environment.

7.4.4 Physical Modeling: The Easel Panel

The large panel in the lower left of Figure 7.3 is an *easel panel*, invoked to edit a target volume. The easel has several parts. The left side is for editing attributes such as the color in which contours are displayed and the Z coordinate of the current slice. Across the top is the *filmstrip*. Each *frame* presents a slice of the model. The image for the slice is in the background. Contours for organs, tumors, and targets are overlaid. The filmstrip provides a scrollable menu for selecting slices for editing. The area in the middle is the canvas, on which geometric editing occurs.

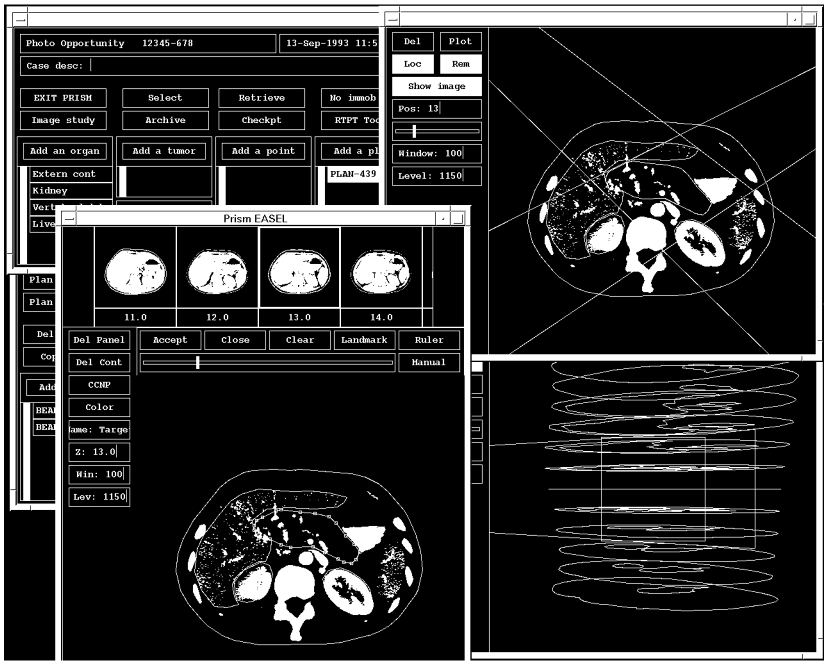


Figure 7.3: A second Prism screen.

To edit a slice, the user scrolls through the filmstrip to find the desired slice 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 displayed on the canvas. All contours defined for any organ, tumor, or target for that slice are displayed but only the one for the entity being edited can be changed—in this case, the target volume, whose contour is outlined with squares to support editing. The user can add, delete, and move these squares to change the target volume contour. The *Clear* button below the filmstrip deletes the entire contour. In addition to a *Manual* drawing mode, the easel supports automatic contouring using a procedure that follows edges on the background image. The easel editing operations actually change a copy of the contour defined in the patient model. If the user presses the *Accept* button, the edited contour is then inserted into the patient model, replacing the old one.

7.4.5 Visualization: Views and View Panels

To support visualization of patient cases (anatomy, tumors, targets, beams, and so forth), Prism allows the user to define any number of *views* of a given plan—orthographic or perspective. Orthographic views come in three kinds, called transverse, coronal, and sagittal, perpendicular to the Z, Y, and X axes of the patient coordinate system, respectively. (See lower right of Figure 7.1.) Prism supports perspective views taken from the origins of given radiation beams. (See lower right of Figure 7.3.)

Each plan panel includes a selector that lists views defined for the plan. When adding a view, the user is queried for the kind of view. Views are persistent, in the sense that they are defined and listed in the selector even when not displayed. Selecting a view from the selector causes the view to be displayed. In particular, selecting a view invokes a *view panel,* which displays the view and allows its parameters to be set. The parameters include the position of viewing plane along the view axis, whether a background radiographic image displayed, the contrast setting for that image, and so on. Each view also displays *locators* to help the user correlate the orientation of the view with the orientations of other defined views. (Locators are discussed further, below.)

Figure 7.1 presents five orthographic views panels. The large one in the upper right is a transverse view. It presents the same slice as in the easel in Figure 7.3, but also displays two radiation beams—as pairs of diverging lines. One beam enters from the upper left; the other, from upper right. The target volume is in the intersection of the beams. The target is imperfectly targeted: the beams hit other organs, including the kidneys (white ovals) and spinal cord (white, irregular shape). This information and the ability to see and change it easily are critical to the treatment plan designer.

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.

Finally, the lower right panel in Figure 7.3 presents a perspective, beam's-eye view, from the perspective of the beam that enters from the upper left in the transverse view, above. This view depicts the sequence of transverse slices through the patient seen from the side. The outer contours represent skin; inner ones, others elements, e.g., organs.

7.4.6 Integration: Easels and Views

Modeling and viewing tools are tightly integrated with each other as they are instantiated. When the user presses *Accept* on an easel, new data is inserted into the model. This causes all graphics to be updated to reflect the changes. Frames in easel film-strips are updated. All affected views are redrawn. If a dose distribution is displayed the data is invalidated and erased from all views in which it appears. In general, changes made to the model cause changes throughout the environment to maintain consistency.

7.4.7 Integration: Views with Views

Visualizing three-dimensional configurations from a set of two-dimensional views is hard. Visual hints that indicate 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 depicting the intersection of the viewing plane of that view with that of some other orthographic view. Locators show how views are oriented relative to each other, which helps the dosimetrist to fuse views into a three-dimensional understanding.

Consider the sagittal view in the lower right of Figure 7.1, for example—a view depicting a slice that divides the patient into left and right parts. The vertical locator in the transverse view corresponds to the viewing plan of this sagittal view. Thus, 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. Horizontal locators in the transverse views depict the intersection of the coronal view (lower left). The vertical line in the sagittal view shows the intersection of the coronal view; and the horizontal locator in the coronal view depicts the intersection of the sagittal view.

Views are thus integrated with each other through locators. Since the viewing planes of views can be changed, and since locators can be dragged, a problem is to keep locators and views consistent with each other. The position of a viewing plane 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 must update. Symmetrically, if a locator is dragged in one view, the viewing plane of the corresponding view changes. Thus, not only do locators provide hints about relative orientations, but they also let user change the parameters of other views. The user can thus "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 complex models—in particular, three-dimensional radiation fields.

7.4.8 Physical Modeling: The Beam Panel

To review, one edits a plan by selecting it using the plan selector on the patient panel, r instantiating a plan panel. For this discussion, the key features of the plan panel are the selectors for radiation beams and views. To add a beam, one uses the add button above the beam selector. To edit a beam, one selects the item for the beam, instantiating a beam panel. On the plan panel on the far left of Figure 7.1, 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 in this case), 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 dial models the position of the gantry that supports the beam apparatus, which rotates on an axis parallel to the floor. Another models the angular position of the collimator, an adjustable masking device through which the beam passes on its way to the patient. The bottom part of the panel displays sliders for adjusting such parameters as the latitude, longitude, and height of the couch on which the patient lies and the shape of the collimator aperture.

7.4.9 Integration: Beam Panels with Beams and Views

Beam panels are integrated with views in that changes to a beam are reflected in views that display the beam. As the gantry dial is moved, for instance, the lines depicting the beam in the transverse views rotate, and the depiction of the patient in the beam's-eye view (from the perspective of the origin of the beam) rotates in three dimensional space. Changes in beam parameters can also invalidate computed dose distributions, causing views to erase depictions of the out-of-date values.

An interesting aspect of the beam panel is its integration with the beam object that it displays. If a user changes the *machine* attribute of a beam ("Clinac 4"), for example, this may imply a change in the kind of collimator used to shape the beam. Different machines have different collimation systems. This, in turn, may require a change in the collimator part of the beam panel: different systems have different parameters, and so require different 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 used. When the user changes the machine attribute, the user interface presented by the beam panel may change to allow editing of a beam shaped by a different kind of collimator.

7.5 Analysis, Specification, Implementation

The architecture of Prism system was conceived, designed, and implemented using behavioral ER modeling and mediators. The rest of this chapter focuses on the use of this method in Prism. This section starts with a simple example to lay a foundation for more complex aspects of the design. After discussing implementation details using this example, I show how the tenets of the approach were used to understand and satisfy the more complex requirements discussed in the preceding section.

7.5.1 The Dialbox Widget

The first example is of a graphical interface widget called a dialbox. The dials in the beam panel (see Figure 7.1) are instances of this kind of "tool"—a tool that can itself be seen as a tiny integrated system. A dialbox is an integrated system in the sense that it integrates two behaviors that can stand alone: a dial and a text line. They are made to work together in the dialbox so that the text and the dial angle remain consistent other as either one is changed—whether by the user or by some other tool.

The behavioral entities of a dialbox are thus 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 programmatic interfaces for getting and setting their values. 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 may be desirable to treat a dialbox not as a system of parts but as a single object. A dialbox is often instantiated, used, and destroyed as a unit, for example. To support the view of the widget as an integral unit, we specify it as an object that has its own operations to get and set an underlying value, the angle displayed in different ways by the dial and text line components.

Behavioral ER Model

When given requirements for a "complex" behavior, such as the dialbox, one begins by asking how it can be decomposed into a system of independent behaviors integrated in a network of separate behavioral relationships. We have no mechanical procedure for finding good decompositions. Rather, we are guided by concerns for independence and integration. What pieces can we use, develop, understand independently; and how should we design them to promote these properties? Given a breakdown into parts, is there a simple behavioral relationship that integrates the parts? What visible structure do we want the clients of the system see?

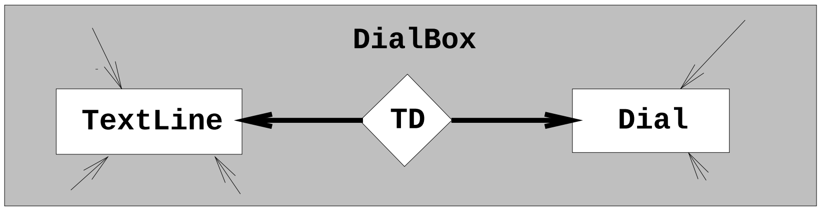


Figure 7.4: The architecture of the dial box “integrated environment.”
The rectangles represent independent entities; the diamond represents the behavioral relationship that integrates them. The dark arrows denote the dependence of the relationship on the entities. The light arrows suggest the entities are visible: used directly by clients.

Reasonable answers for the dialbox are fairly obvious. The system is easily decomposed into a system of two behavioral entities linked by one behavioral relationship, as illustrated in Figure 7.4. The entities are the *dial* and a *text line.* The relationship is simple: if the value stored by one of these objects changes, the value of the other must be updated to reflect the change. This breaks the system into two entities that we can implement, test, and use independently, plus a relationship that is easy to comprehend and implement as a separate unit in its own right.

Mediator Implementation

At the implementation level the task is to represent the entities and relationships as ABT instances while preserving the independence of the constituent behaviors and the separation of integration concerns exhibited in the behavioral ER model. Figure 7.5 presents the essential features of the design of the Prism dialbox widget. The text line and dial are designed and implemented as independent ABTs. The behavioral relationship is designed and implemented as a mediator ABT. The mediator responds to the behavior of each entity (indicated by its event announcements) by updating the other to maintain the specified constraint on the respective values.

In more detail, the dial and text line are designed as ABTs with operations to *Get* and *Set* their values and with events announced when these values change. The events are *NewInfo* and *NewAngle* for the text line and dial ABTs, respectively. The mediator

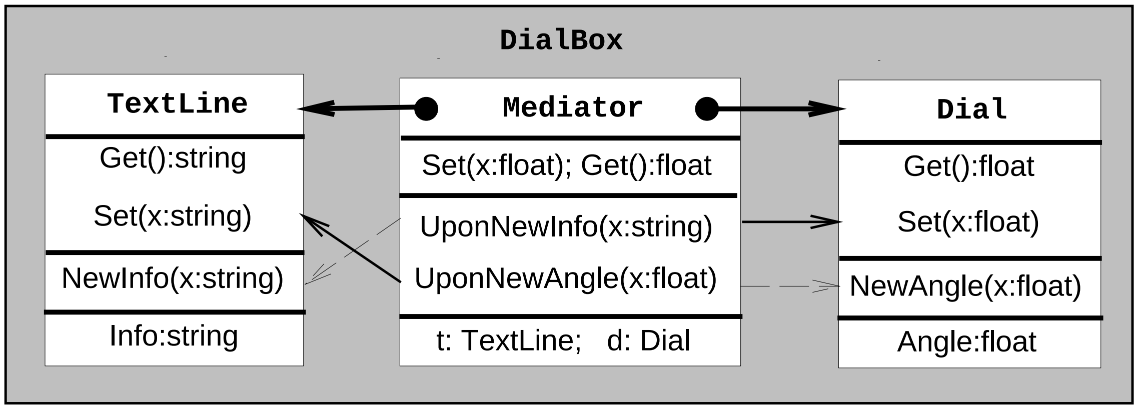


Figure 7.5: Mediator-based design of the dialbox widget.
The heavy arrows emanating from the black dots denote references between objects. The dial and text line are independent; the mediator depends on both. The dashed arrows represent registrations of the operations at the tails of the arrows with the events at the heads. The solid arrows denote explicit invocations of the operations at the heads by the operations at the tails.

references both the dial and text line to register for these events and to call the operations. It registers its *UponNewInfo(x)* operation with the text line *NewInfo(x)* event. When this event is announced, the operation is invoked. The operation computes a new angle a for the dial by converting the updated text (an event parameter) to an angle, and then by calling *Dial.Set(a)* to update the dial with the new angle.

The behavior is symmetrical for changes to the dial. Thus a circularity could result: one update could cause another, and another, etc. There are several ways to break such circularities. One way is for the mediator to maintain a bit that indicates whether an update is already in progress. When invoked, the mediator checks the bit. If not set, the mediator sets it and performs the update. If already set, the mediator just returns. The fact that mediators are objects permits them to maintain such state. Another way to break the circularity would be to define the text line and dial objects as not announcing their events if a new value to be stored is the same as the old.

Low-Level Details

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 quite easy in object-oriented languages, which already support objects with states and operations. We represent events in interfaces as instance variables holding event objects. An event object maintains a list of operations to be implicitly invoked, and the objects to which they are to be applied, when the event is announced [Sullivan and Notkin 92].

Events. The implementation environment for Prism is provided by 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.

Such a class defines operations to *Register* and *Unregister* operations (and the objects to which they should be applied), and another to *Announce* a given event. The mechanism is simple. Source code is presented in Figure 7.6. 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.

Entities. Implementing the *dial* and *text line* object specified in Figure 7.5 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

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

by announcing an *angle-changed* event. The problem is in representing and announcing the event. Figure 7.7 presents the dial class declaration and shows how we guarantee that the event is announced when *setf* is used to update the angle attribute using 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 graphic. (I do not discuss graphics any further.)

Mediators. Implementing the mediator is easy. (Figure 7.8.) In this case, the mediator is the dialbox object itself. The system as a whole is represented by the mediator that integrates the parts—see the *defclass* statement in the figure. The *setf* statement in the class initialization routine *make-dialbox* creates these objects and assigns them to their slots in the dialbox object. This routine also registers the mediator operations with the dial and text line events. The first operation, *upon-angle-changed,* handles changes to the dial; the second, *upon-info-changed,* handled updates to the text. In the registration statements (e.g., *ev:add-notify*) the variable *db* refers to the dialbox (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, which 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.

```
(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 7.7: 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 ds th :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 7.8: Key features of the implementation of the dial/text line mediator.

These mediator's "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 mediator operation uses a busy bit to prevent circularity, as discussed above. Between setting and clearing *busy*, the operation updates the value of the text line and announces the *angle-changed* event of the dialbox (to support clients that treat the dialbox as a unit).

Consider what happens when a client changes the dial angle using the *setf* operation. The wrapper method is invoked by Common Lisp before *setf* executes. First, the dial graphic is erased. The actual setting of the angle value occurs within *call-next-method.* The graphic is redrawn. Finally, the angle-changed event is announced. This invokes the mediator's *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 and, finding an update in progress, returns. Control returns to the text line (its event announcement returns). The text line update completes. Control returns to the mediator. The mediator announces its *angle-changed* event. Finally, the mediator clears the busy bit and returns. The *setf* operation on the dial object completes with the whole dialbox is left in a consistent state.

7.5.2 Multiple Selection List

With the implementation details at the lowest level of Prism described, I now turn to more complex subsystems, starting with the multiple selection lists used within selectors. (A selector includes a multiple selector list as a component.) I will call these multiple selection lists *menus*. A menu displays a list of items. Items can be added and deleted, selected and deselected. When selected, an item is highlighted; when deselected, it is unhighlighted.

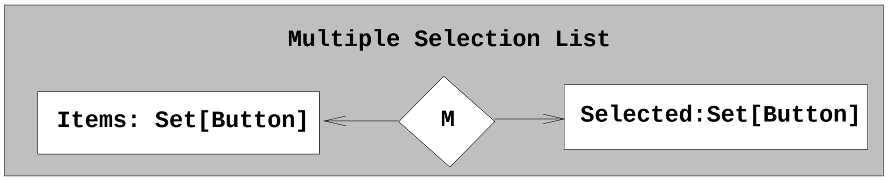


Figure 7.9: Simplified model of a multiple selection list.
A multiple selection list is viewed as a pair of sets of buttons. A button is an item displayed by a menu that can be selected or deselected. The set of buttons called *Items* contains all buttons displayed by the menu. The set, *Selected,* contains 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.

Behavioral ER Model

We designed the menu based on knowledge of how we wanted to use menus in selectors. Within a selector a menu participates in two relationships. The first is a one-to-one correspondence between items in the menu and a set of objects (e.g., the organs of a patient). The second relationship is a one-to-one correspondence between *selected* menu items and active tools (e.g., panels to edit selected organs). If an organ is added to the patient, an item must appear in the list; and if an item representing 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, as a component of the larger selector subsystem, resulted in the model illustrated in Figure 7.9. We view a menu as maintaining two sets of *buttons.* The sets are entities into which one can insert buttons and from which one can delete buttons (more precisely, references to buttons). The buttons, in turn, can be selected or deselected. In standard usage, a client inserts unselected buttons into the *Items* set, then selects and unselects buttons, and then finally deletes them from the *Items* set.

Integration of these sets and the buttons they contain is achieved by the relationship *M* in Figure 7.9. 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—then it must then be inserted into the *Selected* set. Although our implementation does not have this structure, this analysis illuminated the logical structure of a menu, significantly easing the design, implementation, and integration of the menu component.

Implementation

In implementing the menu, we merged the two sets and the behavioral relationship into a single menu object. The menu object thus supports operations to insert, delete, select, and deselect buttons, and events indicating these activities. The select and deselect operations are the equivalents of insertion into and deletion from the selected subset.

We designed the buttons themselves as ABTs with operations for selection and deselection, and events indicating these activities. Selection causes a button to be displayed highlighted; deselection, unhighlighted. The key to integrating buttons with menus is that when a button is inserted into the menu, the menu registers to be notified of the button's selection and deselection events. When the menu is notified of a button being selected, the menu then announces its own *selected* event. This event mimics the one that would have been announced for insertion of the button into the selected subset had the menu been implemented as a pair of sets. (The same thing happens for deselection.) When a button is deleted from the menu, the menu cancels its registrations with the button object.

7.5.3 Selectors

Selectors are used throughout Prism to display the contents of sets that make up patient models (e.g., the organs of a patient), to allow elements to be added to and deleted from these sets, and to allow these elements to be selected for editing. In the face of additions to and deletions from these sets, it is critical to keep the user interface 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 button should updated accordingly; and if the button name is changed, the organ should be updated, symmetrically.

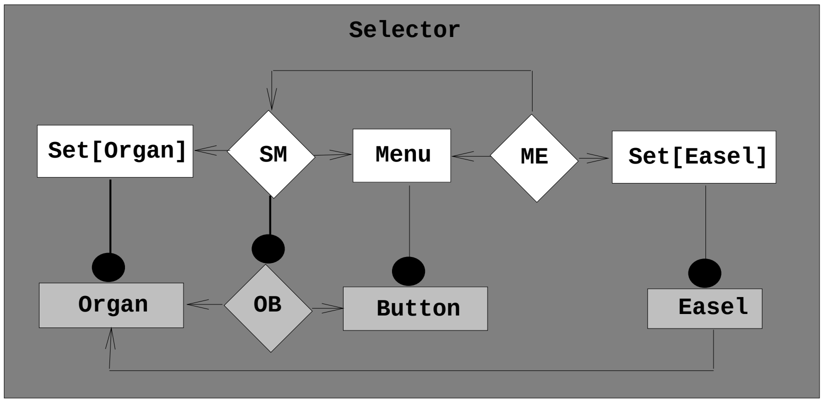


Figure 7.10: Simplified model of the Prism selector subsystem.
A selector in Prism consists of a menu integrated with a set of objects (e.g., organs) to be listed in the menu and a set of tools (e.g., easels), one for each object that is designated by the selection of an item in the menu. The relationship *SM* maintains a bijection between set of objects and items in the menu. The relationship *ME* maintains a bijection between selected items in the menu and tools in the tool set. The relationship *OB* maintains a correspondence between the name of an organ and the name displayed by the corresponding menu item.

Prism eases integration of new tools by making sets, objects, buttons, and so on, directly accessible to tools that need to manipulate the information they represent. Any tool can operate directly on a model, e.g., by adding, changing, and deleting organs, beams, plans, buttons, etc. In the face of such accesses, all aspects of the environment must be kept consistent. I now discuss how we met this requirement, focusing for concreteness on the case of selectors.

Behavioral ER Model

Figure 7.10 presents a behavioral ER model of the organ selector. All other selectors i Prism are exactly analogous. In fact, all of them are instances of the same basic selector class, parameterized at run time to handle sets of different types of objects to be selected (organs, plans, etc) and panels to edit these objects (easels, plan panels, etc.)

The main entities in the behavioral ER model are a set of organs, a set of easels, and a menu. To each set is associated zero or more elements; the number changes dynamically as elements are added and deleted. The two key behavioral relationships, *SM* and *ME,* maintain one-to-one correspondences, as discussed above. If an organ is added to the organ set, *SM* requires a corresponding item (button) to 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 (and thus, in Prism, deleted) by the user, *ME* requires that the corresponding button be deselected.

The relationship *ME* depends on the relationship *SM,* because *ME* is responsible for creating easels for editing organs selected by the designation of menu buttons, but an easel must be connected to the organ associated with the selected button. The association between organs and buttons is maintained by *SM.* Thus, when a button is selected, *ME* then queries *SM* to find out to which organ the new easel should be attached.

The bottom half of Figure 7.10 depicts the individual elements of the sets (organs, buttons, easel panels) and the behavioral relationships between them. *OB* requires and preserves equality of associated organ and button names; if either changes, the other is 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 display part of the easel and the organ displayed into the easel. The reason is that easels are never used in the absence of organs (or generally a volume to be edited). Since the relationship depends on both the easel display and the organ, this merging of integration concerns leaves the easel dependent on the organ.

Implementation

To save space, I discuss only key features of the selector implementation. First, we represent instances of the *OB* relationship as mediator objects. These objects are modeled on the dialbox presented earlier: they propagate values bidirectionally between objects. The relationships *SM* and *ME* are similar to each other, so I will only discuss the implementation of *SM*. I start the discussion with a simple but important part of Prism: the organ set. (Other Prism sets—e.g., panels, tumors, plans—are analogous.)

Sets. We implement the organ set as a *set* 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 parameter 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 (and conversely for deletion).

Representing multi-valued attributes as set ABTs (e.g., the organs in a patient) is one of the cornerstones of the Prism design. Set ABTs provide 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. This is easy because set ABTs announce events that can be observed by mediators.

Designing a mediator to represent *SM* is straightforward. Again, the dialbox provides a template. First, the *SM* mediator holds references the organ set and the menu. Second, it defines four private operations, one to handle each relevant activity, namely, insertion and deletion of organs and buttons. When the mediator is created, it receives references to the set and menu and registers its operations with their events. At runtime, the mediator employs the "busy bit" idiom to avoid circular updates, as in the dialbox.

Relations. In addition, the mediator maintains a relation 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. The relation is implemented as a *relation* ABT, with operations to add, deleted, and look up associations, and with events announcing additions and deletions.

To see how the *SM* mediator works, consider what happens when an organ is deleted. The mediator, having registered to be notified of events signaling deletion of organs, is invoked. It responds by checking its busy bit, which it finds clear (no update in progress).

It maps the organ just deleted through the relation that the mediator itself maintains to find the corresponding button in the menu. It deletes the correspondence from the relation, then calls the menu to delete the button. This invokes the mediator recursively; but now the mediator finds its busy bit set, so it just returns to the menu. Then the menu, having deleted the button, returns to the mediator. The mediator then returns to the organ set. Finally the deletion operation on the organ set 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 key advantage to the method of implementing behavioral relationships as ABT-based mediator objects is that these mediators can be created and can register with other objects dynamically. This is the basis for integrating tools as they are instantiated in Prism.

The idea is that in addition to maintaining a relation between two sets, a mediator also maintains a set of mediators of a different kind responsible for integrating related elements of the sets. Here, *SM* creates an instance of an *OB* mediator to integrate each associated organ and button pair. So, when *SM* adds an association to its relation, it also creates an *OB* instance, providing it with references to the organ and button objects. The *OB* mediator registers dynamically with the events of those objects and keeps them consistent thereafter. When the *SM* mediator deletes the association between the organ and the button, it also deletes the corresponding *OB* mediator. Before being destroyed, the *OB* mediator unregisters from the events of the organ and button objects.

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.

7.5.4 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, I discuss how we handled locators. Recall (see Section 7.4.7) 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 each 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 other view that 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 a locator in one of those views, the Z position of the 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, iso-dose contours depicting dose distributions, etc.

Behavioral ER Model

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 and that we needed to maintain a relation encoding intersections between views in that set. Rather than a one-to-one correspondence between sets, we needed a behavioral relationship that maintains the *intersects* relation as the set of views changes, or as individual views change.

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

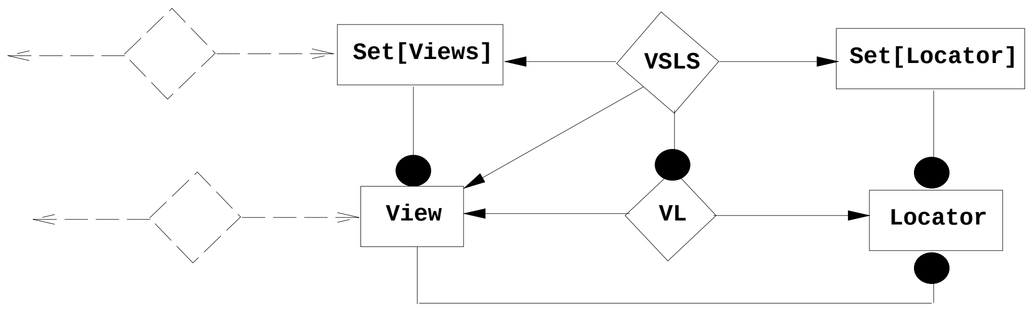


Figure 7.11: The Prism *locators* subsystem.
The locators subsystem contains a set of views and a set of locators. The set of views holds zero or more view instances; similarly for the locator set. The sets are integrated by a behavioral relationship *VSLS* (for view-set/locator-set). *VSLS* keeps the 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.

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 basically consists of a set of views and a set of locators, with locators in one-to-one correspondence with tuples in the symmetric intersects relation. As views are added and deleted, the 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 $L_{A,B}$ displayed in a view A with the view B that the locator represents.

A key benefit of this architecture is that it enabled conception and implementation of this machinery independently of other relationships in which the views and the set of views participate (as suggested by dashed lines and diamonds in Figure 7.11). I already discussed some of the other relationships: the set of views is related to a menu; the name of each view is related to the name of a button; etc. Accessibility of the views and sets permits the behaviors of these entities to participate in multiple behavioral relationships.

Implementation

The implementation follows from the behavioral ER model. The set of views is implemented as a set ABT. The views are defined as independent objects. We changed the representation of the locator set in the 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 that it displays. Locators are thus distributed about the system. We realized *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 views change, the submediators maintain consistency independently of the machinery that deployed them.

The behavioral ER modeling method not only effectively separated concerns in concept, but also in implementation. 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, for example. The mediator method preserved the most important properties of the behavioral ER models in the implementation: independence of entities and separation of integration concerns.

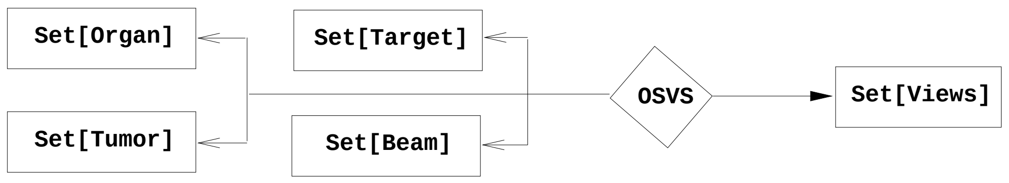


Figure 7.12: The Prism *graphics* subsystem.
The graphics rendering subsystem is analogous to the selector and locator subsystems. The behavior relationship *OSVS* requires that each element in each of the object sets on the left be rendered by a graphic in each of the views in the view set on the right. Although omitted in this figure, corresponding elements are related by submediators, as in the earlier subsystems.

7.5.5 Graphics

Having solved the selectors and locators problems in a collaboration with Kalet, Kalet then independently 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 7.3, 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 easel film strip for the slice being edited , 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 contour graphic representing the particular slice of the target volume. Incremental consistency maintenance is important for acceptable runtime performance, given the capabilities of computer hardware currently on the market.

Figure 7.12 presents Kalets's behavioral ER model for this subsystem. The solution is analogous to those for selectors and locators. In this case the mediator maintains a *cross-product* relation: each element in each subject set (organs, tumors, etc.) must be depicted in each view in the view set. The cross product is between the union of the subject sets and elements depicted in the views in the view set.

The mediator 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 contour. When the user changes the target volume contour in the easel (Figure 7.3), 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 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 wireframe, contoured objects rendered over background, bitmapped 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. Kalet was able to replace 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 made independently 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 separated these concerns.

7.5.6 Beam Panel

Finally, we examine the problem of adjusting the user interface presented by beam panel to accommodate changes in the kind of beam being handled. Consider the beam panel in Figure 7.1. The key aspects of the panel are the button on the left that displays the kind of treatment machine to generate 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.

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 resulting structure seemed arbitrary. Why specialize first by collimator type then by particle type? We tried other orders, too.

We had two problems with the inheritance 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.

Behavioral ER Model

Instead, we defined a single beam class with attributes to indicate specializations of particle type, collimation system, etc. 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 the collimator type of beam and the collimator interface part of the 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 panel responds by replacing both the collimator sub-panel and the "submediator" responsible for integrating the sub-panel with the new collimator object. This is just a simple adaptation of concepts presented for the subsystems discussed above.

7.5.7 Wrap-Up

I 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 developed a new way of thinking about systems, emphasizing both independence 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 models to structurally similar designs and implementations. We applied this approach throughout Prism, at all levels of "granularity," and in solving a range of design problems of several different kinds.

7.6 Development Effort

As of November of 1993, Prism was implemented in about 18,000 lines of Common Lisp [Steele 90] and CLOS [Bobrow et al. 88] and 4,500 lines of Pascal with 11,000 lines of $\LaTeX$ documentation.[2] The Lisp code handles modeling, visualization, and database management. The Pascal, adapted from an earlier system, computes dose distributions. Of the Lisp, about 4200 lines handle user interface widgets, sets, relations, and events. Prism classes take 2700 lines; file handling, 1000; panels, 5200; views and other graphical renderings, 3100; mediators, 1300 (except for mediators that link panels to objects, which

[2] As of August, 1994, the code size has increased to about 35,000 lines, owing mostly to the integration of an AI-based tool for computing target volumes.

are counted with panels); and other code, 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, which is still in use as Prism is phased in, has 41,000 lines of Pascal, 5000 of which are for dose distributions. Comparing with another system, the basic functions taking 18,000 lines in Prism take about 60,000 lines 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[3]

Prism performs adequately on high-end workstations. Kalet uses HP9000 series 700 workstations for development and production. The costliest operations by far are rendering with background images. Background image display can be turned off in a given panel for faster response. Updates to interface widgets—e.g., menus, buttons—are fast. Event registration, unregistration, and announcement are performed many times, but the cost in memory and CPU is negligible in comparison with other functions. The size and number of images in some real cases has exhausted physical memory (128 megabytes) leading to thrashing behavior that degrades performance noticeably.

We built Prism on a modest budget in both money and person-hours, with a small project team. The effort lasted from January 1990 to present. Ten people were involved at different times. The total effort as of November, 1993 was about 4.5 person years. Of this, requirements specification (done before collaboration on design began) took 24 person months. Design and implementation, the focus of the collaboration, took 20 person months. The electron beam dose calculation code took 11 person months.

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.

[3] Personal communication between Ira Kalet and Gregg Tracton of the Department of Radiation Oncology at the University of North Carolina.

Chapter 8

Evaluation

This chapter evaluates how well I have done in this dissertation in solving an important software engineering research problem. I do this by exploring the boundaries of the problem I addressed and the solution I presented. I also raise challenges to this work and either refute them or admit to actual shortcomings.

8.1 Scaling Up

A key concern about any software design method is whether it retains its utility as the scale of problems to which it is applied increases. It is fair to be skeptical about the utility for large systems of methods not yet proven on large projects. Since I cannot justify claims of profitability of the mediator method for large systems on the ground of experience, does not the mediator method therefore fail to pass the key test of scalability? There are several ways to answer this.

First, I have demonstrated scalability of the method to serious but not immensely complex, systems, including commercial products and software that is used in clinical practice in hospitals. These experiences, although not set up as scientific experiments, nevertheless provide valuable anecdotal information validating the claim of profitability of the mediator method for real, useful software systems.

Second, there are no obvious reasons that the mediator method cannot be used for larger systems. I do not believe that behavioral ER models with hundreds or thousands of nodes will be manageable, but rather am encouraged by the fact that behavioral ER modeling is applicable at various scales. Decompositions appropriate for "human consumption" are necessary at every scale. A key property of mediators is that they be recursively structured to integrate behaviors at different scales, from simple switches to more complex types to substantial user-level applications.

Third, success in providing value does not demand demonstrations that behavioral ER modeling and mediator design are profitable when applied to systems that are large in an absolute sense, but only that the approach is *better* than other available methods. I have argued extensively that common methods necessarily scale poorly as demands for integration increase, and that the mediator method overcomes the problems for a many kinds of integration requirements. Moreover, because the mediator method is upwardly compatible with common methods, it is at least as good as they are in other dimensions. Therefore, since little or nothing is lost and something of value is gained the mediator approach is better.

Finally, a dissertation presents the results of a small research project. It is not reasonable to demand nor possible for a single researcher to fulfill demands for large-scale demonstrations (e.g., use on a project involving tens of people over many years). Nor are such demonstrations necessary for valuable progress. I cannot express this idea better than Tony Hoare did a decade ago in discussing formal methods, when he said,

> Most of the books and articles on programming methods are of necessity illustrated only by small examples. Indeed, many of the programming methods advocated by the authors have never yet been applied to large programs. This is not a defect of their research; it is a necessity. All advances in engineering are tested first on small-scale models, in wave tanks, or in wind tunnels. Without models, the research would be prohibitively expensive, and progress would be correspondingly slow [Hoare 84, p. 282].

8.2 Limitations of Implicit Invocation

A behavioral ER model represents a complex behavior as a composition of simpler behaviors in a system of relationships. Each relationship must be satisfied after completion of an operation on a constituent behavior. Behavioral ER models thus compose systems by logical conjunction. Each constituent behavior must satisfy not only its own specification, but also those imposed by each relationship in which the behavior participates.

Mediator based implementations, by contrast, are imperative constructs. Their foundation is broadcast-based implicit invocation, which construct implements sequential invocation (in some unspecified order) of operations registered by mediators representing behavioral relationships. It is well known that logical conjunction cannot always be implemented directly by sequential composition [Hoare 87]. Yet, that is what the mediator method does. Events invoke multiple mediators to implement the conjunction of the behavioral relationships that the mediators implement individually.

The reason that this method works in the examples above is that in all of those cases, the behavioral relationships are independent of each other. In this case, conjunction is implementable by sequential composition—in any order. Satisfying each relationship individually is enough to restore the consistency of the whole system.

The mediator method cannot be applied directly when the relationships to be satisfied depend on each other. If satisfaction of one depends on the satisfaction of another, then mediators implementing the relationships have to be invoked in a particular order to implement logical conjunction. If relationships are mutually dependent there may be no sequential composition corresponding mediators that implements the conjunction.

Do these "shortcomings" undermine the mediator method? The answer is no. First, in many systems independent relationships seems to be the norm. Multiple views can be updated independently, as long as all are consistent at some point, for example. Second, even when relationships are dependent mediators can be defined to maintain *systems* of relationships. Third, enriching the implicit invocation mechanism to support the partial ordering of notifications is another possibility.

8.3 Concurrency, Distribution, Asynchrony, Etc.

It is old-fashioned for software engineering research not to address the daunting problems posed by requirements for concurrency, distribution, temporality, and so on. These are indeed areas in which solutions are needed to ease the "software crisis." How, then, can one positively evaluate this work, which neglects these topics? Although this concern is legitimate, I do not believe this shortcoming undermines this work.

One reason is that sequential behaviors are critical in practice. This dissertation shows that common design methods cannot handle this easy case very well. A contribution to the engineering of sequential systems can have an enormous impact on actual software engineering practice. That is the objective of this work.

Second, solving problems involved in designing sequential systems may ease the design of behaviorally more complex (e.g., concurrent) systems. The unnecessary complexity injected by common design methods into sequential parts of such systems can only exacerbate problems created by concurrency and other such factors. However, I admit that as of now I cannot reliably estimate the impact that the mediator method can have on reducing the cost of behaviorally more complex systems.

Third, it may be profitably to apply the principles underlying this work, primarily the separation of integration concerns, in designing concurrent, distributed, and other such systems. Mediators may be useful to manage synchronization relationships, for example. However, results remain to be demonstrated. I discuss this briefly in Chapter 10.

Fourth, in some cases mediators can be used directly in concurrent, distributed systems. Field is distributed and concurrent, and it uses implicit and explicit invocation, but is basically without rich mechanisms for controlling concurrent access to data. Mediator modularizations of Field-like environments are quite plausible. There may also be valuable solutions that combine the optimistic style of Field with more rigorous concurrency control where needed. Nested transactions [Moss 87] in particular may be valuable in this regard.

8.4 Non-Conservative Integration

The behavioral relationships discussed so far in this work have a common property: they conserve the behaviors they relate. An editor integrated in a behavioral relationship with a compiler can still be used as an editor by its users. In the system of switches, the individual switches can still be turned on and off. The behaviors are extended by behavioral relationships but not restricted.

Another important class of behavioral relationships are non-conservative relationships. These relationships inhibit or restrict the behaviors they relate. The automotive example (see Section 2.1) provides an example of a non-conservative behavioral relationship: between the ignition interlock switch and the ignition switch. Engaging the transmission turns on the ignition interlock, and that prevents the ignition from being turned on; so, the car cannot be turned on with the transmission engaged.

The abstract evolutionary scenario can easily be extended to model this kind of relationship. Suppose the ignition were modeled as a switch $S4$. Recall that the ignition interlock is modeled by $B3$. The new behavioral relationship R that integrates $B3$ and $B4$ strengthens the condition required for $B4$ to be turned on. Not only must $B4$ be off (by the fact that $B4$ is a switch), but R implicitly adds the requirement that $B3$ (the interlock) also be off.

Whereas this dissertation mostly addresses relationships that extend the *postconditions* that specify individual behaviors, non-conservative relationships extend the *preconditions.* This issue raises two questions. Can non-conservative relationships be captured in the modeling framework? Can these relationships be implemented as mediators?

The *integrated* relations are appropriate for modeling non-conservative relationships. One could model R (above) as an element of this relation. However, *affects* is not appropriate for modeling the realization of such relationships. The *affects* relation models behavioral cause and effect. We say $B1$ affects $B2$ because turning on $B1$ implies that $B2$ is also turned on. Extending the modeling framework to model behavioral inhibition is plausible, but beyond the scope of this work.

As to whether non-conservative relationships can be represent by mediators, the answer appears is yes, at least for simple cases. Operations announce events before they execute their main functions. These events provide the opportunity for mediators to intervene to prevent operations from carrying out their functions. Mediators registered to be invoked by these events could check whether it is alright for the operations to occur, and raise exceptions to abort operation if not. A mediator implementing R, when notified of an attempt to turn on the ignition, could check the interlock and raise an exception if the interlock is on, for example. I have not yet tried to validate the usefulness of this approach in practice.

8.5 The Modeling Framework

The validity of the modeling framework itself is a serious concern. There are important aspects of integrated systems that it does not capture well. My main claim for the framework is that it isolates a set of issues and models them well enough to explain problems with common design methods and to suggest a solution. Nevertheless, it is useful to clarify limitations of the framework.

One shortcoming is in the axiom stating that for one module to directly affect another requires one of the modules to reference the other. In actuality, some implementation frameworks allow third party registration, in which a "third party" object C registers an operation O of object M with event E of object N. M subsequently affects N without M or N referencing the other—violating a framework axiom.

That the framework does not model this possibility is not fatal. First, the framework axioms could be relaxed to model this situation. Second, third party registration appears not to be common in practice. None of the methods I discuss use it aggressively, nor does it alone solve the problems with common design methods. Nor is it needed to implement mediators. This observation suggests that it may be useful to provide designers with restricted frameworks that preclude the use of third party registration.

Another problem with the framework is that it models behavioral relationships as binary (*integrated*) relations. Yet, not all behavioral relationships are binary, in practice. Behavioral relationships of higher arities are in fact quite common. This is not a big problem. Although it may be possible to extend the framework to model n-ary relationships, it is not clear that doing so would provide value. I defined the framework first to show how common design methods handle even simple (binary) cases poorly; and second to provide a basis for searching for a solution to this problem. The framework suggested mediators as an abstract solution. The actual mediator construct, being based on first-class objects, accommodates n-ary relationships quite easily. With this practical solution in hand, it is not clear that we need to go back to fix the abstract framework.

8.6 Subtype Polymorphism

Another objection to this work is that it ignores a central aspect of object-oriented design methods: subtype polymorphism. Polymorphism provides the ability to substitute instances of behaviorally extended types where instances of restricted supertype are indicated. In particular, one could place the behavioral extensions needed to integrate objects of one type with other objects in the definition of a subtype of the given type. Neither the supertype nor clients expecting supertype instances would have to be aware of the subtype. Doesn't this solve the problems with common design methods based on ADTs, without the effort of behavioral ER modeling and mediator design?

Subtype polymorphism does not solve the problem. First, subtyping does not lead to externalized, cohesive representations of behavioral relationships. Second, subtyping co-locates representations of behavioral relationships with the representations of the behaviors to be integrated. Third, the subtype approach may require distributing the representations of complex behavioral relationships over multiple subtypes. Fourth, a new subtype may be needed for each subset of behavioral relationships in which a given behavior may participate. This leads to a combinatorial proliferation of subtypes. Fifth, in practice most subtypes are defined statically, so using subtypes to integrate objects dynamically (during execution or evolution) is untenable. Sixth, most subtype concepts do not accommodate what I have called non-conservative integration; rather, subtypes conserve the behaviors of their supertypes. This is understandable given that substitutability of instances is a key objective; but, in practice, designers need to think about non-conservative integration, as well.

Subtyping just does not seem to be the right way to think about integration. A switch $B1$ that works with a switch $B2$ is not clearly best viewed as a different *kind of* switch, but simply as a switch that happens to be integrated with other switches in a system. Nor does multiple inheritance really help. Although a subtype that inherits twice from a *Switch* class does provide a centralized location for the code to integrate two switch instances in one relationship, this structure does not accommodate particpation of objects in changing sets of relationships. Even in the simple case of two switches, the operations of one or both switches will have to be renamed so as not to conflict in the subtype. Indeed, multiple inheritance closely resembles the encapsulation method with respect to the criteria employed in this work. It does not avoid the unneccessary complexity of integration and evolution characteristic of common design approaches.

8.7 Reasoning in the Presence of Implicit Extension

Common software engineering wisdom requires avoidance of side effects. Lehman states "There should be no side effects, no incidental consequences of the execution of any code sequence [Lehman 80, p. 420]." Yet, side effects are key to the mediator method. Calling an operation executes its code, but also codes registered by arbitrary mediator objects. Indeed, it is impossible to specify the runtime effects of an ABT-based operation in the absence of knowledge of the context in which the object is embedded, since meanings can be implicitly extended arbitrarily. Implicit extension compromises denotationality—a serious matter. Does it disqualify the mediator method as valuable for software design?

The answer is no, at least for a broad class of systems. The advantages of being able to integrate components often outweigh the problems caused by having the meanings of components dependent on context. Moreover, any method that uses implicit extension (e.g., MVC, Field) has the same basic property; so, in principle, mediators are no worse than many other methods in this regard. In practice, the aggressive use of implicit invocation in mediator-based designs may exacerbate the problem, however, by increasing the difficulty of debugging, for example. (See also the comments in Section 6.2.6.)

The issue of denotationality does raise interesting research questions. Although the meaning of a component may not be determined locally, it is determined in any given context by combining the local specification with extensions implicitly conjoined by other components. Techniques to help reason about meanings of objects *in context* may help. Transitively conjoining to the specification of a behavior in a behavioral ER model all predicates conjoined by behavioral relationships incident on that behavior yields a specification of the global meaning of the component in context. The analogous approach for imperative implementations is harder.

8.8 Information Hiding

Another possible objection to this work is that behavioral ER models and ABTs compromise information hiding: behavioral ER models by exposing the internal structures of compositions; ABTs by the dynamic activities of objects through event announcements.

Does revealing the structure of a system as a set of behaviors in a network of relationships violate information hiding? The answer in many cases is no. Structure is revealed. Systems are not presented as black boxes. But the structure that is exposed does not necessarily reveal inessential design decisions. Many behaviors in the world are abstractly organized as systems structured as sets of individually visible, behaviorally integrated parts. Think of a car. It is not a black box but an assembly of directly accessed parts: the radio, the transmission, the ignition switch, and so forth. Behavioral ER modeling provide an appropriate means for abstractly modeling integrated structures.

Does implicit invocation violate information hiding by exposing transitions objects undergo? Information is certainly revealed, but the suppression of inessential design decisions is not compromised. The visible behaviors in which objects engage are, in the ABT view, essential properties of objects. They are not implementation details. ABTs simply support a different class of abstractions than ADTs. Whereas ADTs models machines with actuators, ABTs model machines with both actuators and indicators (events). ABTs do not reveal the "internal wiring" implementing their abstract interfaces.

8.9 Alternatives to Abstract Behavioral Types

I chose ABTs for implementing behavioral ER models because they provide a general building block patterned on the familiar ADT and powerful enough to support mediator design. The key feature of the ABT is that it makes events dual to operations. Events names and signatures are declared, and events have precise semantics. Hoare suggests what designers need are abstractions that support clear thinking about problems and solutions, and providing a basis for notations supporting clear expression of such thoughts [Hoare 68]. I believe ABTs satisfy this requirement.

However, one might ask why adopt the ABT given many existing frameworks that support implicit invocation: Smalltalk-80, Field, etc? Indeed, the use of the mediator approach does not strictly require ABTs. One could implement behavioral relationships as separate tools using the Field or Smalltalk-80 implicit invocation mechanisms. What I find is that ABTs helps designers structure their thoughts by focusing attention on audible behaviors as a key, abstract property of objects. To develop a sense for advantages and shortcomings of ABTs, I now discuss them in the context of related constructs.

Smalltalk-80 Change Notification. Smalltalk-80 does not declare events; nor do objects register operations with events. Rather, objects register themselves as dependents of other objects. This effectively registers fixed operations defined in the *Object* base class with events also defined there. (The details are a bit more subtle, as discussed elsewhere [Sullivan and Notkin 92].) One specializes the events by passing application-specific values as event parameters, usually indicating that some "aspect" of an object changed. One specializes registered operations by redefining them in subclasses of *Object.*

This approach seems to encourage designers to think in terms of traditional ADT-like objects that depend directly on each other in certain, standard ways. The "update" dependence of views on models is the archetypal design idiom promoted by this kind of framework. In contrast, ABTs support a more symmetrical abstraction. Every object potentially announces and responds to explicitly declared events.

Events in the Global Scope. Another possibility is to organize events as belonging not to individual objects but to a "global scope," in the way that operations do not belong to individual objects in the Generalized Object Model [Bellcore 92]. Some frameworks, such as Field, do just this. The message server manages a single, large space of events; and tools really register for these global events.

There are several reasons to prefer ABTs. Perhaps the most important is that a global event approach is a more radical departure from common design methods. The cost of making the transition to ABTs is already substantial, even for some experienced designers, as discussed below. Since a shift to global operations and events it is not necessary to solve the problems that I have identified, the case for that more extreme departure from familiar patters is not warranted.

I do not mean to suggest that there are not other problems that justify such a departure; but those problems are not addressed by this work. If, however, it becomes clear that global events, external to any particular objects, are indicated for a particular design, the ABTs can still be used to advantage. It is easy to simulate global events using ABTs. One need only define an ABT having a single event in its interface, and instantiate an object of that type in the global scope. This does require that objects be able to announce events other than those in their own interfaces, which has not been necessary for the designs discussed earlier.

8.10 Anticipating Event Interfaces

A common challenge to the ABTs is that it requires designers to anticipate the events needed in an object's interface—and, implicitly, that this is unreasonably hard. It is true that the only behavioral relationships that can be represented by mediators are those expressible in terms of events and operations declared in the interfaces of the objects to be integrated. It is true that if an event or operation needed by a mediator is missing, then the desired relationship cannot be represented by a mediator without interface changes.

This is not a problem that can be solved, nor is it unique to the mediator method or to ABTs. Object interfaces generally have to be rich enough to support clients. If they are not, they have to change—or implementations can be changed, circumventing interfaces altogether. The mediator method drives designers to make required interfaces explicit. If a needed event *or operation* is missing, it has to be added. In practice, based on the experiences reported in this work, such changes are not frequent nor costly; nor does adding events to an interface compromise the independence of objects.

A related problem, encountered only once (by C. Brett Bennett) is that event interfaces grow quite large if many different behaviors have to be made audible; and in this case it may not be desirable to implement each one as an event object (as discussed in Chapter 7). In the case where this problem arose, Bennett basically fell back on the Smalltalk-80 style, by defining one "physical" event parameterized in various ways to represent different but related "logical" events. However, this is truly but an implementation detail. The approach to designing objects with arbitrary, designer-specified event interfaces was strictly followed.

8.11 Does the Mediator Method Ease Design?

A hard question to answer is whether the mediator method makes people better software designers. In my experience, it does, although this does not happen without effort. It is hard for people to make the “paradigm shift” to the new design method. It was hard for me, in retrospect; and it appears to be hard for most everyone. Once the shift is made, though, leverage is obtained, as it becomes easier to design useful decompositions of complex systems. At some point, the designer sees that integration and independence are reconciled—that one can design parts locally yet have them work globally.

To make the shift, one has to get a feel for behavioral ER modeling, for the models that can be implemented directly using ABTs, and for the care that has to be taken in precisely defining the semantics of mediators, and for the kinds of transformations that takes models that cannot be directly implemented using ABTs to ones that can. Systems of mutually dependent relationships, in particular, have to be recognized as not directly implementable by sets of corresponding mediators. One has to know that such a system can be replaced by single, more complex relationship that handles the system as a whole. The more complex relationship can then be implemented as a mediator—albeit a more complex one. Once one has acquired proficiency with these ideas (which in my experience seems to take the effort of building one or two complete systems), the benefits surpass the costs and the positive value of the mediator method becomes apparent.

Chapter 9

Connections to Related Work

Ezra Pound has said, "You can't judge any chemical's action merely by putting it with more of itself. To know it, you have got to know its limits, both what it is and what it is not. What substances are harder or softer, what more resilient, what more compact [Pound 37]." In this spirit, this chapter characterizes the work presented in this document by placing it in the context of other related efforts.

9.1 Behavioral Entity Relationship Modeling

In this section, I consider work related to behavioral ER modeling, in particular, emphasizing the conception, specification, implementation, and evolution of relationships separately from the entities related. The idea of relationships as constructs distinct from related entities is centuries old, of course. It is a central idea to modern mathematics. And, more to the point, it has been exploited in software design for many years. This section relates behavioral ER modeling to other software engineering efforts in which the separation of relationship concerns figures prominently.

9.1.1 Entity-Relationship Data Modeling

Separating the representations of relationships from those of the entities to be related in software engineering dates at least to Chen's entity relationship (ER) data modeling method [Chen 76]. Before Chen, data modelers tended to represent relationships in terms of attributes (fields) of the entities (relations) to be related. To represent the employment of an employee by a company, an employee record would contain a field holding the identifier of the record for the corresponding company in the company relation. This is analogous to what I have called hardwiring. Chen's approach was to represent the relationship as a separate relation. Thus, *entity relations* model entities, such as employees or companies, while *relationship relations* model relationships, such as the employment of employees by companies.

The hardwiring of relationships into entities caused the same sorts of problems for data models that it causes for behavioral models: clarity is reduced as the simplicity and independence of the entity relations is compromised, and evolution is unnecessarily complicated by the need for entities to change when relationships are added to, deleted from, or changed in a data model. Chen's separation of relationships helped solve these problems and provided the software engineer with a better way to think about not only data models but application domains themselves.

The separate representation of behavioral relationships in behavioral ER models is clearly based on the same basic idea as ER data modeling. The value added by *behavioral* ER modeling is not in the general idea but in the richer kinds of behaviors that are represented in this way. Data modeling cannot capture the rich behaviors needed to implement complex, integrated systems, such as Prism. Data modeling can express some behavior, but not enough. Cardinality constraints on relationships, for example—e.g., every employee has one boss—do imply some behavior: e.g., adding an employee requires adding an association mapping the employee to a corresponding boss [Kilov and Ross 94]. However, simple constraints of this kind are not rich enough to effectively model the systems described in this work.

9.1.2 The Object-Relationship Model

Rumbaugh [Rumbaugh 87] is largely responsible for introducing the relationship as first-class concept to the object-oriented community. He argued that representing relationships in terms of pointers in the objects to be related reduces clarity and complicates software evolution. This notion is the object-oriented analog of Chen's idea.

To solve the problem, Rumbaugh proposed that relationships be represented separately, using constructs distinct from the classes used to represent objects. He also proposed to given relationships behaviors by allowing their definitions to specify that operations should "propagate through them." For example, a *part-of* relationship, associating the wheels of cars with the car, might be annotated to propagate the deletion operation: then deleting the car would also delete its wheels. This idea was incorporated into a popular object-oriented modeling and design method [Rumbaugh et al. 91].

Behavioral ER modeling, although similar in spirit, differs both in its emphasis on defining relationships with rich behaviors and in its approach to implementing behavioral relationships as mediators. The difference in behaviors can be seen by comparing the association between a car and its wheels to the bijection mediator that incrementally maintains a relation that encodes a one-to-one mapping between two sets. Allowing relationships to have complex behaviors eases the design of complex systems. Behavioral relationships are a more general-purpose modeling construct than Rumbaugh's associations.

The mediator method also employs a single basic building block, the ABT, in contrast to the separate class and relationship constructs advocated by Rumbaugh. The addition of events to interfaces enables use of this one construct to represent both behaviors and relationships. This uniformity has at the important advantage that mediators can be treated as objects by other objects. The dialbox mediator discussed in Chapter 7 is an example. There is no fundamental need for a separate construct, and some important advantages follow from using the single ABT construct.

9.1.3 Contracts.

Helm *et al.* make the idea of generalized behavioral relationships explicit in a paper sketching a language for specifying such relationships [Helm, Holland, and Gangopadhyay 90]. The key construct in this language is the *contract.* A contract specifies a relationship by defining the roles of the relationship (e.g., employee and employer), the interfaces that participating objects must support, preconditions on the objects for participation in the relationship, a procedure for initializing the objects in a relationship, an invariant over the states of the objects, and invocation obligations on the objects to maintain the invariant. To promote the design of reusable relationships, contracts are generic with respect to participating objects. A construct called a *conformance declaration* adds a level of indirection to allows particular objects to participate in generic relationships. These constructs defines how objects discharge the obligations imposed by contracts.

This work shares with behavioral ER modeling the basic view that behavioral relationships should be treated as first-class constructs; but it differs in important ways. First, the language is intended for specification, not implementation, while ABTs provide a medium for implementing relationships in common programming languages. The ISO Generalized Relationship Model [ISO/IEC JTC1/SC21/WG4 92, Kilov et al. 92] is similar in this regard. Second, contracts often specify that objects invoke each other. This biases implementations towards hardwired designs. Third, contracts emphasize relationships over the objects they relate: "the specification of a class becomes spread over a number of contracts and conformance declarations, and is not localized to one class definition [Helm, Holland, and Gangopadhyay 90, p.178]." Yet contracts are justified by the idea that in other languages, "behavioral compositions ... are spread across many class definitions [Helm, Holland, and Gangopadhyay 90, p.169]." The mediator method, by contrast, emphasizes equally relationships and the objects they relate. Finally, the behavioral ER modeling method does not emphasize genericity of relationships, although it does not preclude it. It is not hard to define *adapter* objects to add the level of indirection needed for generic mediators.

9.1.4 Constraint Programming

Constraint programming systems are also based on a separate representation of relationships. Such systems include Penguims [Hudson and Mohamed 90], ThingLab II [Freeman-Benson, Maloney, and Borning 90], Kaleidoscope [Freeman-Benson 90, Lopez, Freeman-Benson & Borning 93] (which extends the ThingLab II mechanisms by adding linguistic constructs for abstracting low-level constraints into compound constraint objects), and CLP($\mathcal{R}$) [Jaffar *et al.* 92]. Using these systems, one organizes a program as a set of variables and a set of constraints over those values that are to be solved, or maintained as the variable values are changed.

A key technology behind these systems is a global representation of the constraints and variables in a system. Such a representation provides the basis for automatic handling of conflicts and for efficient constraint satisfaction. This automation relieves designers from having to handle such tasks manually. The declarative notations used in some constraint systems also spare designers from having to translate specifications into error-prone imperative code.

In exchange for these advantages, constraint programming systems often restrict the set of problems that can be handled. Some system only solve systems of linear inequations [Jaffar *et al.* 92]. Others loosen the semantic restrictions while retaining automation, but at the cost of having to give up declarative notations. ThingLab II [Freeman-Benson, Maloney, and Borning 90] strikes an interesting compromise in this dimension. It promotes design in terms of logical constraints and variables; and it implements constraints and variables as corresponding physical "constraint" and "variable" objects that are imperatively progrmmed. To this extent, the mediator method is remarkably similar to ThingLab II. Neither approach relieves designers of the task of writing imperative code, but both encourage effective entity relationship-based decompositions of that code.

The difference between the two approaches is that ThingLab II accepts certain semantic and structural restrictions in return for the ability to provide certain benefits automatically. The mediator method dispenses with such mechanisms, placing more responsibility on the designer, but in return provides the designer with an unencumbered, fully general, implementation framework. The benefits to designers of complex, integrated environments far outweigh the cost of not having the automated consistency checking that ThingLab II provides. In all fairness, the developers of ThingLab II do not promote it as a platform for complex, integrated environments, but as a system to support graphical user interface development.

To understand this tradeoff, one must look inside ThingLab II. A ThingLab II "variable" is an imperatively programmed object that stores a *value* of an arbitrary type. A ThingLab II "constraint" is an imperatively programmed object that implements a logical constraint. In particular, a "constraint" is an object exporting a set of procedures, one for each constrained "variable." Each procedure satisfies the logical constraint by setting the value of the "variable" to which it is associated, based on the values of the other constrained "variables." The system tracks the "constraints" and "variables" in a program. The system uses this information in several ways. First, given an indication that the user is going to change a particular variable, the system selects a set of procedures—one from each "constraint" connected directly or indirectly to that variable. These procedures are composed sequentially into a program whose execution resatisfies of all the logical constraints after any change in the value of the designated variable. The system also uses its representation of the global constraint graph to detect and flag prohibited structures in the imperative code—value cycles in particular.

This implementation structure imposes severe semantic and structural restrictions that make this system inappropriate for implementing complex, integrated environments. One problem is that "constraints" compute values in a functional manner, which makes impractical incremental consistency maintenance in the face of small changes to structured objects, such as sets [Maloney 91]. Another problem is that the constraint satisfaction procedures cannot update more than one constrained variable; but doing this is critical for incremental constraint maintenance. The bijection mediator updates one set and its own relation in the face of insertions on another set, for example.

9.2 Mediators

Just as separating relationships is an established software engineering concept, so is implicit invocation; even the implicitly invoked mediator is not an unprecedented construct, although it way not previously developed in the general way presented in this dissertation. This section discusses the mediator method as presented in this work in relation to previous efforts in which mediator-like constructs have appeared.

9.2.1 AP5.

AP5 [Cohen 89] is a declarative, constraint-based programming environment based on Common Lisp [Bobrow et al. 88]. AP5 extends Common Lisp with a transactional, relational database supporting *triggers* (procedures invoked when specified conditions are satisfied). A trigger is a pair $t = (p, m)$, with a declaratively specified predicate p, and an arbitrary imperative action m. The predicate is expressed in a notation based on first-order logic extended with relational operators and with temporal operators for naming the state of the before and after commitment of a proposed transaction.

When a transaction is proposed, the system reevaluates all predicates in the context of the transaction. The system invokes the action parts of triggers whose predicates are satisfied in that context, in which the future state of the database reflects commitment of the transaction. To maintain a constraint, one defines a trigger whose predicate reflects the condition that would reflect a violation of the desired constraint. If proposed transactions would violate the constraint, the action part is invoked. It can then either modify and resubmit the transaction, abort it, or take some other appropriate action.

Strengths of AP5 include incremental satisfaction of predicates as transactions are processed and support of atomic transactions. Incremental satisfaction is important for efficiency in a database with triggers. It is too costly to reevaluate all predicates on each transaction. Transactional semantics are important for maintaining global consistency. It should be possible to abort a change and all of its effects if consistency cannot be maintained.

The comparison of AP5 and the mediator approach is interesting. First, the AP5 mechanism (and, implicitly, design method) revolves around a central, relational database. In contrast, behavioral ER modeling does not depend on any centralized mechanism or structure. In fact, it leads to systems with decentralized, network-like structures. Second, AP5 uses powerful mechanisms for predicate matching and transactions. These mechanisms have obvious benefits; but they also carry costs. Designers must acquire and learn and commit to a new environment, design method, and language. By contrast, the cost for an everyday software designer to adopt the mediator method is much lower. The method does not seriously constrain the choice of programming language or environment, nor does it require sophisticated and possibly costly mechanisms. Bennett's geographic information browser was written in Objective C and runs on DOS. The mediator method provides more value to the average designer of modest systems.

However, the mediator's lack of support for transactions (owing primarily to the lack of support in common languages) is a serious shortcoming for developers of large or complex systems. It appears that mediators could exploit transactions if they were implemented on a platform supporting objects and nested transactions [Moss 87]; but I still have yet to verify the practical utility of this combination.

9.2.2 APPL/A

APPL/A [Sutton, Heimbigner, and Osterweil 90] is an Ada-based programming language that supports software process programming. The extensions to Ada include relations with programmable implementations, triggers that respond to operations on relations, optionally enforceable predicates on relations, and statements with transaction-like capabilities. APPL/A handles concurrency by allowing the designer to specify the synchronization of event announcement with the execution of the implicitly invoked methods. APPL/A also allows components that receive multiple event notifications to prioritize them. Notifications are queued at components, which then handle the events in priority order.

Of particular interest in this work is the APPL/A implicit invocation mechanism. In contrast to the ABT, which permits objects to export and announce designer-specified events, only relation objects, instances of a special APPL/A type constructor, can announce events. Moreover, the events that relations can announce are restricted to insert, delete, update, and find—indicating invocation of these relation operations.

These restrictions imply that the programmer must either use explicit invocation mechanisms to integrate objects that are not relations, or model the objects as relations. If explicit invocation is used, the benefits of implicit invocation, such as easier evolution, are lost. If all components are modeled as relations, then the programmer is prevented from using more natural representations where appropriate.

APPL/A can be viewed as supporting a special case of the ABT. The mediator method aggressively generalizes the special case, providing the designer with a more versatile medium for conceiving and representing systems. The APPL/A system provides a model for concurrency control lacking in the mediator approach. Marrying the strengths of these efforts is a promising research direction.

9.2.3 Chiron

Chiron-1 [Keller, Cameron, Taylor, and Troup 91] also exhibits what can be seen as a special case of the mediator method. In Chiron-1, *artist* objects maintain consistency between arbitrary objects and objects called *abstract depictions,* representations from which a concrete, graphical displays of the underlying objects can readily be generated. An artist updates an abstract depiction when the artist is implicitly invoked by an event that indicates invocation of an operation on the underlying object. The system announces these events automatically, relieving the designer from having to declare or announce. In the reverse direction, part of the system called the *abstract depiction manager* may invoke artists explicitly to change the underlying objects.

The "active data types" of Chiron-1 differ from ABTs in that the designer of an ABT explicitly declares and implements events and programs event announcements. Chiron-1 relieves the designer of this task, at the cost of restricting events announced to those that indicate operation invocations. If that is the only kind of event desired, the Chiron-1 approach provides a greater value to the programmer than the ABT, since it provides the same benefit at a lower cost. (Chiron-1 handles concurrent updates properly.)

In my experience, other events are useful often enough that the flexibility provided by ABTs is valuable, despite the cost to declare, implement, and announce them. A simple example is a set ABT announces the insertion of an element to a set only if the insertion operation actually changes the set, which is not necessarily on every invocation of the insertion operation. Making events as general as operations really changes the way designers can think about system components.

9.2.4 Views for Tools

Garlan defined yet another mediator-like construct: the compatibility map. In an integrated environment, different tools may be designed to operate on the same information in different ways. The problem of the consistency of different views of common information arises. Garlan handled the problem by representing views as ADTs and by *merging* views that represent the same information [Garlan 87]. In merging, the separate interfaces of distinct ADTs are preserved, but multiple implementations are replaced with a single implementation that supports both the views individually as well as consistency among the views.

The system produces merged implementations automatically by selecting appropriate *compatibility maps.* These mediator-like constructs are selected inductively based on the system's knowledge of the basic types and relationships available. After merging, different interfaces on the shared information cab be used by different tools. A compiler may view data as a *set* interface, while another tool views it as a *list,* for example. In Garlan's approach a system has a fixed set of compatibility maps and composition rules; follow-on work loosened this restriction by providing a language for defining maps and compositions [Habermann et al. 88].

An advantage of this approach is that it allows merged implementations to be optimized, with potentially significant savings in both time and space. Two views containing identical elements may share a single instance of that element in a merged view. On the other hand, the approach prevents types from being integrated simultaneously in several relationships: a view interface can be bound to only one merged implementation. Nor is merging desirable in all cases: protection or reuse concerns may suggest separate implementations, for example. Finally, in practice, generating merged implementations may be costly, since the types and relationships needed for a broad range of environments are not likely to be known by the system in advance.

The mediator approach dynamically integrates type instances, and so passes up the opportunity for optimization of integrated representations. There is the theoretical possibility of runtime optimization, but that remains a hypothetical that may be addressed in the future. On the other hand, the flexibility and low cost of mediation using ABTs provides an approach that is available to programmers today, that has been shown to have significant value in several ambitious development efforts, and that admits the network structures of behavioral ER models, in contrast to the highly restrictive structures of types statically integrated through compatibility mappings.

9.3 Implicit Invocation

The use of implicit invocation in software design has a long history. Many programming languages, systems, and methods have included or been based on mechanisms of this kind [Goldberg and Robson 83, Habermann and Notkin 86, Stefik, Bobrow, and Kahn 86, Krasner and Pope 88, Cohen 89, Reiss 90, Cagan 90, Sutton, Heimbigner, and Osterweil 90, Collins et al. 91, Gorlick 91, Harrison, Kavianpour, and Ossher 92] However, a unified view of the behavior, benefits, and design space for such mechanisms emerged only recently [Garlan and Notkin 91, Sullivan and Notkin 92]. A detailed discussion of this work is beyond the scope of this dissertation. I refer the interested reader to the literature.

9.4 Structure and Evolution

Finally, this dissertation is based on a large body of work on the relationship between the structure of software representations and key software engineering outcomes, including ease of design and evolution. Dijkstra suggests that perhaps, "the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution[Dijkstra 72, p.124]." The contribution of this work in this dimension is to develop a nice way to factor the behaviors of integrated systems and their corresponding software representations.

The important works of Belady and Lehman [Lehman and Belady 85] characterize statistical invariants that appear to govern the development, evolution, value over time, and ultimate ossification of large software systems. In these terms, the contributions of this work are, first, a characterization of how using common methods to design integrated systems unnecessarily hastens the decay that robs systems of their value; and, second, a new method that, when carefully applied, can significantly delay the onset of decay and thus increase and extend the value of integrated systems.

In contrast to the statistical—essentially thermodynamical—approach of Lehman and Belady, David Parnas takes a more classical approach to characterizing relationships between structure and evolution. Parnas's foundational work on information hiding [Parnas 72] argues that not all modular decompositions of software representations are equally good with respect to ease of evolution; but rather that evolution is eased by decompositions in which modules hide design decisions that are subject to change—in particular, choices about data representations and algorithms. Those modularizations are best that anticipate likely changes. In this regard, the contribution of this work is a way of modularizing systems, at both the conceptual and implementation levels, that anticipates changes in intergation requirements: in particular, the addition, deletion, and modification of behaviors and behavioral relationships in integrated systems.

Parnas's Work on extension and contraction is also relevant [Parnas 79]. Here, Parnas argues that cyclical dependencies among modules complicates the design and evolution of families of systems, the reason being that all modules must be present and work properly for any one to do so. To make this idea more precise, Parnas models systems in terms of modules related in a *uses* relation. He argues that, absent other concerns, the designer should strive for a hierarchical *uses* relation. In this case, modules at lower levels remain independent of those at higher levels, allowing one to more easily deliver subsets and supersets of a system's functions.

The modeling of representational structure in terms of modules and relations in this work clearly has a precedent in Parnas's paper on extension and contraction. The mediator method can also be analyzed in terms of Parnas's *uses* relation. The key to the mediator method is that modules that implement behaviors do not *use* (or call or reference in any other way) modules outside of themselves, but that modules that implement behavioral relationships inherently do *use* (and reference) the modules they relate. The mediator method thus naturally leads to a hierarchical *uses* (and references) structure. This property obviously cannot guarantee freedom from other problems (such as multiple mediators using the same behavioral objects in conflicting ways), but it does keep behavioral modules independent, in the sense of *uses,* from modules with which they are behaviorally integrated.

Chapter 10

Conclusion

In this dissertation, I characterized common software design methods and showed how they significantly and unnecessarily complicate the design, realization, and evolution of integrated systems. I presented the mediator method as a solution. This method combines behavioral ER modeling as a design technique, with an approach to implementing behavioral ER models in terms of ABTs. The mediator method yields substantial benefits at low cost to provide significant value to practicing software engineers. I substantiated this claim both by analytic reasoning and by careful reflection on experiences with the method, especially the design of the Prism radiation treatment planning system.

Despite the simplifications made by the models and the qualitative character of the experiences, I believe I have adequately defended my thesis: *the mediator method is significantly better than common software design methods for designing, realizing, and evolving synchronous, sequential integrated systems.* The mediator method not only allows engineers to design more effectively, but—by providing broader, more tightly integrated, more easily adapted behaviors than otherwise feasible—enables the delivery of more effective systems to users. These systems then provide added value to their beneficiaries, e.g., by enabling better cancer treatment. That benefits society, which then, in turn, supports works such as this one. This ideal, closed cycle provides for an incremental, monotonic increase in gross value, which must be our ultimate goal.

The reality is that investments do not always pay off, but the potential for gain requires risk. Nor does one winning bet suffice for all time. The value of any fixed technology decreases over time. The problem is to win often and big enough to sustain the loop of increasing value. Thus, the key question at the end of a winning effort is *what next?* To conclude this dissertation, I therefore sketch future efforts that appear to stand a reasonable chance of adding value by extending the results of this work.

A formal semantics for behavioral ER modeling. The problem is to *precisely* specify behaviors—and in a form that eases design, integration, evolution. We need formal specification languages that admit behavioral ER modeling as a design strategy and architectural style. We therefore need a formal (e.g., denotational) semantics for such languages. There are some serious research problems here. One is that implicit extension, which is at the heart of behavioral ER modeling, appears to conflict with denotationality—since the behavior of an expression can be implicitly extended, depending on the context in which it is embedded. One solution is to accept local non-denotationality while providing automatic support for synthesizing denotational expressions giving the meanings of expressions in the context of behavioral ER models.

A formal semantics of abstract behavioral types. The problem is to precisely specify the behavior of ABTs in a form that is minimally biased to a particular implementation. Heterogeneous algebras provide a basis for formalizing abstract data types. Is there an analog for abstract behavioral types; or are ABTs really just a notational device for denoting behaviors that can already be formalized as abstract data types? If the latter, how are ABTs formalized, and what is the mapping to the heterogeneous algebraic specification of equivalent ADTs?

Language support for behavioral ER modeling and ABTs The primary problem is to get people to think in terms of behavioral ER models and mediators. The secondary problem is that the software languages we use significantly influence the ways we think;

and that common languages do not encourage, and may even discourage or preclude, thinking in terms of behavioral ER models and ABTs.

There is thus a need to investigate languages to support the interlocking activities of thinking and expressing in terms of behavioral ER models and ABTs. We need formal specification languages and, perhaps implementation languages even more. Programming language semantics is an important issue here: typing issues; scope rules; efficient compilation of efficient code; event parameter passing and return; exceptions; transactions; concurrency.

Empirical data on software evolution. The problem is to better understand the problem of the evolution of integrated software systems. How do the behaviors tend to evolve; how do the software representations respond; and how does the choice of software representation limit the adaptability of behaviors in practical terms? We do not have a great deal of empirical data on this topic. The solution is to study reality. I propose to carefully track and reflect upon the evolution of a real system built using the techniques in this work: Prism. The specific question is how much of the evolution of the system falls into categories identified in this work: adding, deleting, and changing behaviors and behavioral relationships; to what extent does the system architecture localize and ease or distribute and complicate changes; in particular, what desired behavioral changes are unexpectedly hard, and why?

Scaling up in complexity of behaviors. The primary problem is that many real-world integrated systems are not synchronous, sequential, but rather involve many asynchronous, concurrent threads of control. Computer-aided design environments supporting large design teams are obvious examples. Integration and evolution problems are just as critical as in the systems addressed by this work, and are even harder to solve owing to the combination of larger scale and the need to maintain consistency in the face of concurrent accesses. I propose to decouple scale and concurrency, and to study concurrency in particular. This dissertation provides a precedent: the solution that worked for the small-scale switch example scaled up to work just as well for Prism. I propose to add concurrency to the mix and to study techniques that extend the mediator method as presented here to handle problems in this more complex behavioral domain (and similarly for temporality and non-conservative integration).

Other work. There are many other research directions to take based on this work. I conclude by briefly mentioning a few: developing tools for behavioral ER modeling and for the synthesis and analysis of mediator implementations; organizing development teams around the behavioral ER model structure; and reverse engineering and re-engineering based on the recovery or synthesis of behavioral ER models of the behaviors of existing software systems.

Bibliography

[Bellcore 92] Bell Communications Research, "The Framework: A Disciplined Approach to Analysis," *Science and Technology Series ST-OPT-002008,* Issue 1, May, 1992.

[Bennington 56] H.D. Bennington, "Production of Large Computer Programs," *Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers,* June 1956, pp. 15–27. Also in *Annals of the History of Computing,* October, 1983, and *Proceedings of the 9th International Conference on Software Engineering,* (Computer Society Press), 1987.

[Birrell and Nelson 84] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Call," *ACM Transactions on Computer Systems 2,*1, pp. 39–59, February, 1984.

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

[Boehm 88] Boehm, B.W. "A Spiral Model of Software Development and Enhancement," *Computer,* May, 1988, pp. 61–72, in P.W. Oman and T.G. Lewis, Eds., *Milestones in Software Evolution,* (Los Alamitos: IEEE Computer Society Press), 1990, pp. 249–260.

[Brooks 86] F.P. Brooks Jr., "No Silver Bullet—Essence and Accidents of Software Engineering," it Information Processing 86, H.J. Kugler, Ed., (North Holland), 1986, pp. 1069–1076, in *Milestones in Software Evolution,* P.W. Oman, Ed., (Los Alamitos: IEEE Computer Society Press), 1990, pp. 293–300. Also appears in *Computer,* April 1987, pp. 10–19.

[Brooks ???] F.P. Brooks Jr., *Academic Careers for Experimental Computer Scientists.*

[Cagan 90] M.R. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal, 41,*3, pp. 36–47, June 1990.

[Cameron 89] Cameron, J.R., *JSP and JSD : The Jackson Approach to Software Development,* (Washington : IEEE Computer Society Press), c. 1989.

[Chase *et al.* 94] J. Chase *et al.*,, On the Opal system, *ACM Transactions on Computer Systems,* forthcoming in 1994.

[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.

[Cohen 89] D. Cohen, "Compiling Complex Transition Database Triggers," *Proceedings of the 1989 ACM SIGMOD,* 1989, Portland, Oregon, pp. 225–34.

[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.

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

[DeMarco 79] T. DeMarco, *Structured Analysis and System Specification,* (Englewood Cliffs, NJ: Prentice-Hall), 1979.

[Dijkstra 65] E. Disjkstra, "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.

[Dijkstra 72] E. Disjkstra, "The Humble Programmer," ACM Turning Award Lecture, ACM Annual Conference, Boston, August 14, 1972, in E.N. Yourdon, Ed., *Classics in Software Engineering,* (New York: Yourdon Press), 1979, pp. 113–125.

[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.

[Freeman-Benson, Maloney, and Borning 90] B. Freeman-Benson, J. Maloney, A. Borning, "An Incremental Constraint Solver," *Communications of the ACM 33,*1, pp. 54–63, January, 1990.

[Freeman-Benson 90] B.N. Freeman-Benson, "Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming," *Proceedings of OOPSLA/ECOOP 90,* 1990, Ottawa, Canada, pp. 77–88.

[Garlan 87] D. Garlan. *Views for Tools in Integrated Environments.* Ph.D. Thesis, Carnegie-Mellon University, 1987.

[Garlan and Ilias 90] D. Garlan and E. Ilias, "Low-cost, Adaptable Tool Integration Policies for Integrated Environments," *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments,* 1990, Irvine, California, pp. 1–10.

[Garlan and Notkin 91] D. Garlan, D. Notkin, Formalizing Design Spaces: Implicit Invocation Mechanisms. VDM '91, Formal Software Development Methods. Appears as Springer-Verlag Lecture Notes in Computer Science #551 (November 1991).

[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.

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

[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.

[Gorlick 91] Gorlick, M. M. and Razouk, R. R., "Using Weaves for Software Construction and Analysis," *Proceedings of the 13th International Conference on Software Engineering,* Austin, Texas, May, 1991, pp. 23–34.

[Guibas and Stolfi 85] L. Guibas and J. Stolfi, "Primitives for the Manipulation of Three-Dimensional Subdivisions," *ACM Transactions on Graphics 4,*2, pp. 74–123, April, 1985.

[Habermann and Notkin 86] A. N. Habermann and D. Notkin, "Gandalf Software Development Environments," *IEEE Transactions on Software Engineering SE-12,*12, pp. 1117–1127, December 1986.

[Habermann et al. 88] A.N. Habermann, C. Krueger, B. Pierce, B. Staudt, and J. Wenn. Programming with Views. Technical Report CMU-CS-87-177, Carnegie-Mellon University, January, 1988.

[Harrison, Kavianpour, and Ossher 92] W. Harrison, M. Kavianpour, and H. Ossher. Integrating Coarse-Grained and Fine-Grained Tool Integration. IBM Research Division, Research Report RC 17524 (#77482), January 8, 1992.

[Helm, Holland, and Gangopadhyay 90] R. Helm, I.M. Holland, D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *Proceedings of OOPSLA/ECOOP 90,* 1990, pp. 169–180.

[Hoare 68] C.A.R. Hoare, "Record Handling," in F. Genuys, ed., *Programming Languages,* pp. 291–347, Academic Press, 1968.

[Hoare 84] C.A.R. Hoare, "Programming: Sorcery or Science," *IEEE Software 1,*2, April, 1983, pp. 5–16, in P.W. Oman and T.G. Lewis, Eds., (Los Alamitos: IEEE Computer Society Press), 1990, pp. 273–283.

[Hoare 87] C.A.R. Hoare, "An Overview of Some Formal Methods for Program Design," *IEEE Computer 20,*9, 1987.

[Hudson and Mohamed 90] S.E. Hudson and S.P. Mohamed, "Interactive Specification of Flexible User Interface Displays," *ACM Transactions on Information Systems 8,*3, pp. 269-288, 1990.

[ISO/IEC JTC1/SC21/WG4 92] ISO/IEC JTC1/SC21/WG4, "General Relationship Model—Third Working Draft: ISO/IEC JTC1/SC21 N 7126," May, 1992.

[Jackson 75] M.A. Jackson, *Principles of Program Design,* (London: Academic Press), 1975.

[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.

[Jaffar *et al.* 92] Jaffar, J., Michaylov, S., Stuckey, P., Yap, R. "The CLP($\mathcal{R}$) Language and System," *ACM Transactions on Programming Languages and Systems 14,*3, July, 1992, pp. 339–395.

[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.

[Keller, Cameron, Taylor, and Troup 91] R.K. Keller, M. Cameron, R.N. Taylor, and D.B. Troup, "User Interface Development and Software Environments: The Chiron-1 System," *Proceedings of the 13th International Conference on Software Engineering,* May, 1991, Austin, Texas, pp. 208–218.

[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.*

[Kilov and Ross 94] Kilov, H. and Ross, J., *Information Modeling: an Object-Oriented Approach,* (Englewood Cliffs, N.J.: Prentice Hall), 1994.

[Kilov et al. 92] H. Kilov, B. Moore, L. S. Redmann, "Proposed U.S. Comments on General Relationship Model—Third Working Draft," Accredited Standards Committee X3—Information Processing Systems Document Number X3T5/92–296 X3T5.4/92–1142, September 14, 1992.

[Krasner and Pope 88] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object Oriented Programming 1,*3, pp. 26–49, August/September 1988.

[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.

[Larus 89] J. R. Larus, *Restructuring Symbolic Programs for COncurrent Execution on Multiprocessors,* Ph.D. dissertation, UC Berkeley Computer Science, May 1989. Also appears as Technical Report No. UCB/CSD 89/502.

[Lehman 80] M.M. Lehman, "Programs, Life Cycles and Laws of Software Evolution," *Proceedings of the IEEE Special Issue on Software Engineering 68,* 9, September, 1980, pp. 1060–1076, in M.M Lehman and L.A. Belady, Eds., *Program Evolution: Processes of Software Change,* (London: Academic Press), 1985, pp. 393–449.

[Lehman 81] M.M. Lehman, "Programming Productivity—A Life Cycle Concept," *Proceedings of CompCon 81,* IEEE Cat. No. 81CH–1702–0, September, 1981, pp. 232–241, in M.M Lehman and L.A. Belady, Eds., *Program Evolution: Processes of Software Change,* (London: Academic Press), 1985, pp. 469–489.

[Lehman and Belady 85] M.M Lehman and L.A. Belady, Eds., *Program Evolution: Processes of Software Change,* (London: Academic Press), 1985, pp. 355–373.

[Linton, Vlissides, and Calder 89] M.A. Linton, J.M Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews," *Computer 22,*2, pp. 8–22, February 1989.

[Liskov and Zilles 75] B.H. Liskov and S. N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions of Software Engineering SE-1,*1, March, 1975, pp. 7–19.

[Lopez, Freeman-Benson & Borning 93] Lopez, G., Freeman-Benson, B. and Borning, A., “Kaleidoscope: A Constraint Imperative Programming Language,” in Mayoh, B. *et al.* (Eds.), *Constraint Programming,* NATO Advanced Institute Series, Series F: Computer and System Sciences, Springer-Verlag, (1993). Also appears as Technical Report 93–09–04, University of Washginton Department of Computer Science, Seattle, Washgington, September, 1993.

[Maloney 91] J. Maloney, *Using Constraints for User Interface Construction,* Ph.D. Dissertation, University of Washington Department of Computer Science and Engineering Technical Report 91–08–12, August, 1991.

[McCabe 91] T. McCabe. Programming with Mediators: Developing a Graphical Mesh Environment. Masters Thesis, University of Washington. 1991.

[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.

[Moss 87] J. Moss, “Nested Transactions: An Introduction,” in *Concurrency Control and Reliability in Distributed Systems.* B. Bhargava, ed., Van Nostrand Reinhold, 1987.

[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).

[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.

[Parnas 79] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering SE-5*,2, pp. 128–138, March, 1979.

[Pound 37] Ezra Pound, *The ABC of Reading,* (New York : New Directions), 1960, c. 1934 (1987 printing).

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

[Ritchie and Thompson 78] D.M. Ritchie, K. Thompson, "The Unix Time-sharing System," *Bell System Technical Journal 57*,6, part 2, pp. 1905–1930, July-August, 1987.

[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.

[Royce 70] W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of Wescon,* August, 1970, also in *Proceedings of the 9th International Conference on Software Engineering,* (Computer Society Press), 1987.

[Rumbaugh 87] J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language," Proceedings of OOPSLA, 1987, pp. 466–481.

[Rumbaugh et al. 91] J. Rumbaugh, . Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modeling and Design,* (Englewood Cliffs: Prentice Hall), 1991.

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

[Snyder 89] L. Snyder, "The XYZ Abstraction Levels of Poker-like Languages," *Proceedings of the Second Workshop on Parallel Compilers and Algorithms,* 1989, Urbana, Illinois.

[Spivey 89] J.M. Spivey, *The Z Notation: A Reference Manual,* (Prentiss Hall International), 1989.

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

[Stefik, Bobrow, and Kahn 86] M.J. Stefik, D.G. Bobrow, and K.M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," *IEEE Software,* pp. 10–18, January, 1986.

[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 Methodology 1,* 3, July, 1992.

[Sullivan, Kalet and Notkin 93] K. Sullivan, I.J. Kalet, and D. Notkin, "Prism: A Case Study in Behavioral Entity-Relationship Modeling," University of Washington Department of Computer Science Technical Report 93-09-03, September, 1993.

[Sutton, Heimbigner, and Osterweil 90] S. Sutton, D. Heimbigner, and L. Osterweil, "Language Constructs for Managing Change in Process-Centered Environments," *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments,* 1990, Irvine, California, pp. 206–17.

[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.

[Yourdon 78] E. Yourdon and L. L. Constantine, *Structured Design,* (New York: Yourdon Press), 1978.

[Zurcher and Randell 68] F.W. Zurcher and B. Randell, "Iterative Multi-Level Modeling—A Methodology for Computer System Design," IBM Res. Div. Rep. RC–1938, Nov. 1967. Also Proc. IFIP Congr, 1968, Edinburgh, Aug. 1968, pp. 138–142 (Ch 1,3,19,20,21).

Vita

Kevin J. Sullivan

Birth: December, 29, 1960, Oxnard, CA

Citizenship: USA

1442 Westwood Road

Charlottesville, Virginia, 22903 USA

(804) 295-7756

Fix phones

Department of Computer Science

Thornton Hall

University of Virginia

Charlottesville, Virginia, USA 22903

Tel. (804) 924-7605

Internet: sullivan@cs.virginia.edu

Education

University of Washington, September 1987 to August, 1994

Ph.D. Computer Science and Engineering (1994)

Dissertation: *Mediators: Easing the Design and Evolution of Integrated Systems*

Advisor: Professor David Notkin

M.S. Computer Science (1990)

Topic: Formal Specification of Software Systems

Tufts University, USA, September 1979 to May, 1987

B.A. Mathematics and Computer Science, (1987)

Employment

September 1994–, Assistant Professor, *Department of Computer Science, University of Virginia,* Charlottesville, Virginia, USA

January 1994–March 1994, Teaching Assistant, *Department of Computer Science* *and Engineering,* Seattle, Washington, USA

April 1988–June 1993, Research Associate, *Department of Computer Science and Engineering,* Seattle, Washington, USA

June 1990–September 1990, Visiting Faculty Associate, *Tokyo Institute of Technology,* Tokyo, Japan

September 1987 to March 1988, Teaching Assistant *Department of Mathematics, University of Washington,* Seattle, Washington, USA

1981–1987, Senior, Middle, Junior Systems Programmer, *Tufts University Computer Services,* Medford, Massachusetts, USA

Honors and Achievements

GTE Graduate Fellowship (1989-1990)

General Electric Thomas Alva Edison Award (1979).

New Hampshire State Debating Champion (1979).

New Hampshire State Extemporaneous Speaking Champion (1978).

Publications

- Sullivan, K., *Reconciling Software Integration and Evolution: Behavioral Entity-Relationship Modeling and Design,* Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Forthcoming.

- K.Sullivan, I.J. Kalet, and D. Notkin, "Prism: A Case Study in Behavioral Entity-Relationships Modeling and Design," Submitted to the 16th International Conference on Software Engineering, Also available as University of Washington Department of Computer Science and Engineering Technical Report 93-09-03 (September 1993).

- Notkin, D., Garlan, D., Griswold, W.G., and Sullivan, K., "Adding Implicit Invocation to Languages: Three Approaches," To appear, *Proceedings of the JSSST*

International Symposium on Object Technologies for Advanced Software (November 1993). (Will appear in a Springer-Verlag Lecture Notes in Computer Science volume.)

- Sullivan, K. "Implicit Extension Enables Behavioral ER Modeling in Formal Specification," 1993, draft, by request.
- Sullivan, K.J. and Notkin, D., "Behavior Abstraction," University of Washington, Department of Computer Science and Engineering Technical Report 92-03-08, March, 1992.
- Sullivan, K.J. and Notkin, D., "Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methodology 1,*3, July, 1992.
- K. Sullivan, "Abstract Behavioral Types for Behavioral ER Design," Workshop on Object-Oriented Reasoning in Information Systems, *Addendum to Proceedings of OOPSLA92,* September, 1992.
- Sullivan, K.J. and Notkin, D., "Behavioral Relationships in Object-Oriented Analysis," TR-91–09–03, University of Washington, Department of Computer Science and Engineering, 1991.
- Sullivan, K.J. and Notkin, D., "Reconciling Environment Integration and Software Evolution," TR-91–08–08, University of Washington, Department of Computer Science and Engineering, 1991.
- Sullivan, K.J. and Notkin, D., "Reconciling Component Independence and Environment Integration," *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments,* 1990, Irvine, CA.
- Sullivan, K.J., Salman, M., and Van Evera, S. "SIOP: A Computerized Nuclear Exchange Model for Civilian Defense Analysts," in Eden, L. and Miller, S., eds., *Nuclear Arguments,* (Ithaca: Cornell University Press), 1989.

- Salman, M., Sullivan, K.J., and Van Evera, S. "Analysis or Propaganda: Measuring American Strategic Nuclear Capability, 1969-1988," in Eden, L. and Miller, S., eds., *Nuclear Arguments,* (Ithaca: Cornell University Press), 1989.

Software Artifacts

- Prism: Co-Modeling and -design, with Ira Kalet, of a tightly integrated, object-oriented, 3-D environment for interactive planning of cancer radiotherapy treatments; UW Department of Radiation Oncology, 1992–1993; scheduled for clinical usage in Fall, 1993. This system is discussed in a number of technical reports, available upon request.

- SLIK, A Simple Lisp Interface Kit: Consultation on modeling and design of a Common Lisp/CLOS-based user interface toolkit. This is described in Kalet, I. and Kromhout-Schiro, "SLIK Programmer's Guide," Technical Report 93–05–01, Radiation Oncology Department, University of Washington, May 21, 1993.

- Various Frameworks: A collection of C++ frameworks for constructing small integrated environments. These include implicit invocation (event notification), collections, n-dimensional affine geometry for graphics, and user interface widgets built on the InterViews system; UW Department of Computer Science and Engineering, (1988–1991).

- CurveToy: An interactive, graphical tool for exploring surface interpolation and approximation schemes; UW Department of Computer Science and Engineering, (1989).

- SIOP: A strategic nuclear exchange model for civilian defense analysts; Center for Science and International Affairs, Kennedy School of Government, Harvard University, (1989). Co-published with *Nuclear Arguments,* referenced above.

- Orca Prototype: An integrated programming environment for large-scale, non shared-memory parallel machines; UW Department of Computer Science and Engineering, (1988).
- Various Systems Utilities: e.g., Unix-based user-level tape mount manager, user access management systems; Tufts University, (1981–1987).

Invited Talks

- "Escape from the Abstract Data Type," CHIFOO, Object-Oriented Workshop, Portland, Oregon (1993).
- "Abstract Behavior Types for Behavioral ER Design," Workshop on Object-Oriented Reasoning in Information Systems, OOPSLA, (1992).
- "Integrating Independent Software Components," Tektronix Research Labs, Beaverton, Oregon (1990).