Appears in: *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'99)*
Cocoa Beach, Florida, October 1999

Separating concerns in direct manipulation user interfaces

R. E. Kurt Stirewalt
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824

Abstract

Direct-manipulation user interfaces are difficult to implement as a layered hierarchy. Features, such as drag enabling and continuous graphical feedback, require frequent interaction and collaboration among a large number of objects in multiple layers. These collaborations complicate the design of the interfaces to the various layers. We present a new component-interface model called the mode component whose features simplify the expression of enabling and feedback collaborations across layer boundaries. We illustrate the use of mode components through a large example.

1. Introduction

The *direct-manipulation* style of interaction now dominates desk-top applications and is required for applications such as graphical design editors and interactive simulation. In this style, a user articulates *commands*—user initiated operations that affect the state of the underlying application—by manipulating graphical objects rather than making a sequence of selections or typing text [20]. We are investigating how to design these systems as a hierarchy of software layers, where each layer represents the system at some useful level of abstraction.

Our layering criterion is based on the time honored principle of *model-view separation*, which aims to decouple the user interface (UI) from the domain objects that implement the core functionality of the system. This principle, which was first applied in the SmallTalk Model-View-Controller (MVC) paradigm [11], is motivated by the desire to reuse the domain objects in different applications and requires the domain objects to have no direct knowledge of UI objects. Using this principle, we want to design hierarchical software components that separate domain objects and commands from the graphical objects that represent them. Henceforth, we shall refer to these graphical objects as *interactors*.

Layering is complicated by the need to provide *feedback* before and during a complex interaction. Feedback refers to the reflection of the legality of a command that is articulated by one or more interactors in the visual appearance of those interactors. To support feedback requires a flow of information from the domain objects to the interactors. Such an information flow compromises model-view separation and is generally difficult to accomplish in a layered design. Consequently, the implementation of these features tends to compromise the integrity of the layers.

We developed a new component model, which we call the *mode component model*. In this model, a designer specifies the component's interface using a variant of Harel's StateChart formalism [8, 9]. Like a traditional component model, the interface to a mode component defines a collection of services, each of which can be invoked by a component at a higher level in the layering hierarchy. An interface under the mode-component model, however, also associates temporal and functional pre-conditions with each service and provides a continuously updated view of its current mode. These modes and pre-conditions are observable by higher level components. Moreover, the implementation of a mode component contains primitives for observing and reacting to mode changes in lower-level components.

A mode component describes the behavior of the system at one level in the hierarchy. By stacking one mode component on top of another, mode and service–pre-condition information can flow transparently from lower level components into higher level components. This transparency simplifies the design of feedback behavior without compromising model-view separation. As information flows up through a level, low level modes and services are transformed into higher level modes and services.

The original idea to use a modeling notation to represent a single layer and then treat the stacking of layers as a transformation from one instance of a model into another came from the work on GenVoca [5]. GenVoca components use object models (from OO-design [19]) to represent the interface to each layer, and the implementation of each layer is a transformation of objects and operations at one level into

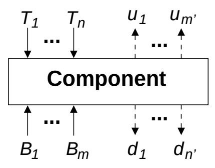


Figure 1. Model of component exteriors [5].

objects and operations at another. Mode components use a variant of StateCharts, which have also been adopted by OO-design methodologies, to describe the behavior of each layer. Moreover, the implementation of each layer is a transformation from modes and events at one level into modes and events at a lower level. This paper introduces the model underpinnings of Mode components. We will discuss how to implement a stack of mode components in Section 5.

2. Background

We designed the mode-component model to balance the requirements of separation of application and UI concerns against the information needs required to implement feedback. Feedback is the reflection of the *legality* of a command in the appearance of one or more interactors that are used in the articulation of the command. We distinguish *enabling*, which is feedback that is provided prior to the beginning of an interaction, from *interim feedback*, which is provided during the course of an interaction. Feedback is vital to the usability of a DMI because the command that an interactor is being used to articulate may be ambiguous prior to and during the interaction. In a graphical file browser, for example, the dragging of an icon might represent either the delete-file command or the move-file command. The ambiguity cannot be resolved until the user drops the icon over either the waste-basket or a folder icon.

2.1. Hierarchical component models

The information needs associated with feedback complicate a layered design. To explain these complications, we adopt the terminology of Batory and O'Malley [5]. A layer is synonymous with a component or a set of components. The *interface* of a component is anything that is visible outside the component; everything else about the component is considered part of its *implementation* [17]. Figure 1 illustrates an hierarchical component. In this model, the T_i are *top* operations, which higher level components can invoke for lower level services. Conversely, the B_i are *bottom* operations, which lower level components can invoke for higher level services. The set $\{T_1, \ldots, T_n, B_1, \ldots, B_m\}$ is the component's interface. The component's implementation may invoke operations d_i and u_i, which request services from lower (respectively higher) level components.

Because feedback is concerned with the legality of a command, information about this legality must flow from the underlying application layer into the UI layer. Figure 1 suggests that information flow can be implemented in one of two ways: Either the application layer provides services that the implementation of the UI layer invokes, or the implementation of the application layer can invoke bottom services in the UI layer that notify interactors when a command's legality changes. The former mode of information flow tends to be used to implement interim feedback; whereas the latter tends to be used to implement enabling.

For a UI component to evaluate the legality of a command in an application component, the interface of the application component must provide services that can be invoked to perform the evaluation. Moreover, these services must support evaluation over a *partial set* of the command's inputs. Consider, for example, the command move-file in the afore-mentioned file browser. This command takes two parameters, a source file and a target directory. To decide whether or not to enable a given interactor requires computing the legality of move-file given only the source file that corresponds to the given interactor. On the other hand, to compute interim feedback when a given interactor (the source) is being dragged over another interactor (the target) requires evaluating the legality of move-file given both the source and target files. Consequently, the single command, move-file, leaves a footprint of three services in the interface to the component. One service actually invokes the command; whereas the other two services evaluate the legality on one and two arguments respectively. Unfortunately, this duplication makes it difficult to modify commands (unless these modifications do not change the precondition), and the variants are an artifact of the direct manipulation style of interaction rather than a generally useful service in the component. If these additional services could be automatically generated, then the component will be easier to maintain.

Another issue is the notification that is required to enable interactors. When the legality of a command changes, some interactors may need to be notified. The need to maintain this *view consistency* can easily lead to conflicts with the model-view separation principle, which forbids domain objects to be designed with any knowledge of the UI. Fortunately, solutions to this problem are well known. The Chiron environment [28, 24] addresses view consistency by automatically augmenting domain objects with the logic that notifies dependent view objects when a change occurs. Each domain object is modeled as an *abstract data type*, which can be *wrapped* by special listening agents that export the same interface. Clients of a domain object perform operations on the wrapper, which invokes the core operation and then notifies the view. Because the notification is introduced automatically, it does not compromise the reuse of domain objects. Implicit notification is also easy to implement using constraint features, such as are found in the Garnet/Amulet object system [14, 13].

2.2. Specifying feedback

Part of the difficulty in designing components for feedback is specifying the conditions under which a command is legal. These conditions reflect temporal constraints on the global execution of commands as well as any functional pre-conditions on the execution of the command. The other difficulty lies in the specification of how the legality of a command affects the appearance or enabling of an interactor. Unfortunately, each of these specifications and their interplay affects the interface to both the application and UI components.

Abowd and Dix argue that feedback is best specified as an *event-delimited, status constraint* [3]. A status constraint is a formula that derives the status of an interactor as a continuously updated function of the status of a domain object. The constraint is event delimited, in the sense that it only holds during an interval that is bound by the occurrence of events. In [21], we use hierarchical states, such as are provided by the StateChart formalism [8], to represent event-delimited intervals. A StateChart can constrain a transition based on the internal state of another (concurrent) StateChart. So using states to represent event-delimited intervals enables a designer to easily specify event-delimited status constraints.

Trætteberg observes that using a StateChart representation of *gestures*, such as dragging an icon of one type and dropping it over an icon of another type, simplifies the expression of how interactor combinations denote commands [25]. This suggests a nice three-layered approach to specifying DMIs by stacking StateCharts on top of one another. The lowest level layer is represented by a StateChart whose states represent application modes and whose events are commands that affect the domain objects. The middle layer is represented by a StateChart whose states represent gesture states and whose events represent interactor collaborations during an interaction. Finally, the top layer is represented by a state chart whose states represent the observable states of each interactor and whose events represent device events that each interactor services.

2.3. Summary

The information needs required to implement feedback conflict with the desire to separate application and UI concerns into different components with rigid interfaces. Feedback is difficult to design to begin with, and when it is superimposed on a layered design, it leaves an undesirable footprint on the interfaces of application and UI components.

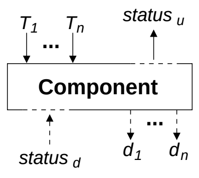


Figure 2. Mode component schema.

The StateChart formalism, however, has features that are useful for specifying feedback. If we can design components whose interfaces have the interactive power of StateCharts, then we can solve the feedback problem without violating the layering principles.

3. Mode components

A *mode component* (MC) is a model of an hierarchical software component whose interface provides a continuously updated view of its current status (Figure 2). The services that can be invoked are the T_i, and the view of the component's mode is represented by $status_u$. The implementation of a mode component may invoke lower level services (the d_i). In addition, a component can directly observe the current mode of a lower level component, and it can observe the *enabledness* of a service with respect to the current mode of a lower level component and one or more parameters. These additional facilities are represented as $status_d$ in the figure. This model admits no bottom operations (B_i) because such operations compromise the critical design separation principles. Rather, every aspect of feedback is expressed entirely in terms of the status, which flows transparently between layers.

3.1. MC features and their use

A change in the status of one MC can *trigger* an event in a higher level MC. Consequently, one component can declare its status to be a continuously updated function of the status of a lower level component. Triggering conditions are quite useful for describing how the visual appearance of an interactor relates to the status of the domain object it represents and its enabling with respect to commands that it might be used to articulate.

There are two forms of status observation. A component can observe whether or not a lower level component is currently *in* one of a set of named states and whether or not a lower level component currently *enables* a service with a given set of parameters. This latter feature is defined to work over an incomplete set of inputs. The interpretation of such a check is that it is possible to perform the service given the information known so far. We use this feature extensively in enabling and feedback constraints that must reflect the legality of a command on only a partial set of data.

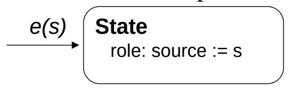
To allow the specification of event-delimited status constraints, a mode can be annotated with auxiliary transient data, which we call *roles*. Constraints are specified through an idiom of defining appropriate roles in a lower level component and then constraining the status of a higher level component on the low-level status introduced by the roles.

3.2. Formal introduction to MCs

We model the interface and implementation of an MC through a disciplined use of UML state diagrams [7]. These diagrams are based on Harel's StateChart notation [8], with special conventions for modeling the state and behavior of objects. We chose UML over standard StateCharts because some of the status relationships that we need to specify refer to the values of attributes in specific objects [1]. Moreover, many practicing software engineers know UML; so MCs can be easily adopted into practice. The discussion that follows includes inline diagrams that use the StateChart notation. Recall that states in a StateChart are specified as roundtangles; states are hierarchical; arrows that connect states represent transitions; and arrows are annotated events subject to a guard.

We restrict the interaction features of StateCharts to enforce the hierarchical composition of MCs, as outlined in Figure 2. The event-broadcast semantics of StateCharts are restricted so that events flow exactly one level down the hierarchy. Specifically, if an action of the form send e is performed in a component at level *i*, then e is multicast to all of the components at level $i-1$. A similar restriction applies to the use of guards and triggers that refer to the internal state of another component. For a condition of the form [C **in** S] to appear in the implementation of an MC at level *i*, C must be a level $i-1$ component, and S must name a state in C.

We extend the StateChart formalism with two concepts: roles that incorporate event-delimited data into the status of a component and quantified enabling conditions. A role increases the amount of status that is visible in the interface of a component. A *role* is a named object whose lifetime is bounded by a state. Roles are introduced through the keyword **role** and are always initialized to some value. A role is said to be filled when the state in which it is declared is entered. In the example below, the State identifies a role source, which is initialized to the parameter s of event e.



[1]A dedicated object constraint language (OCL) is being adopted into UML to handle such conditions [26].

We also extend StateCharts with the **enables** feature, which is a concise shorthand for complex status conditions. The full form of the enabling condition is:

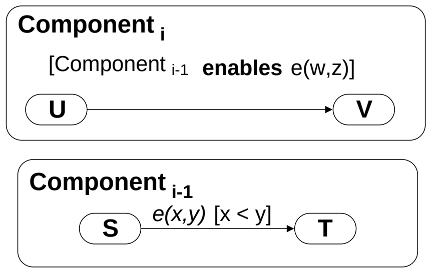
$$\exists \vec{x} \bullet C \textbf{ enables } e(\vec{y}, \vec{x})$$

where C names a component, *e* names an event, $\vec{x}$ denotes a sequence of fresh identifiers, and $\vec{y}$ denotes a sequence of expressions over identifiers that name object attributes and roles. The condition is true if there exist x_i such that component C is receptive to the event $e(\vec{y} \frown \vec{x})$, where the $\frown$ operation is sequence concatenation. To understand this feature, we first describe the simple case in which there is no quantifier.

The condition

$$C \textbf{ enables } e(\vec{y})$$

is true if component *C* is in a state with an outgoing transition on event *e*, and if the inputs in $\vec{y}$ satisfy the precondition of *e* on that transition. The following example shows snippets of the MCs of two components in adjacent layers.



$Component_{i-1}$ can transition from state S into state T upon receiving an event $e(x,y)$ such that $x < y$. The condition on the transition in $Component_i$ names an instance of event *e* with values *w* and *z* as actual parameters. The condition is true if $Component_{i-1}$ can perform a transition upon receiving $e(w,z)$. That is, the condition is true if $Component_{i-1}$ enables, or "is receptive to", the event with the given parameters.

The full form allows some of the event parameters to be existentially quantified. The quantifier notation was motivated by the use of existential quantification to hide variables. Abadi and Lamport noticed that the formula $\exists x \bullet S$ specifies the same system as *S* except with the variable *x* hidden [1]. By existentially quantifying unknown variables, the resulting formula is is optimistic: If any possible configuration of unknown values can make the formula true, then the existentially quantified formula is true. The idea has been adopted into other specification methods, such as the Z schema calculus [27].

3.3. Examples of use

Figure 3 shows a mode-component description of the drag-and-drop gesture. We present it here because we will

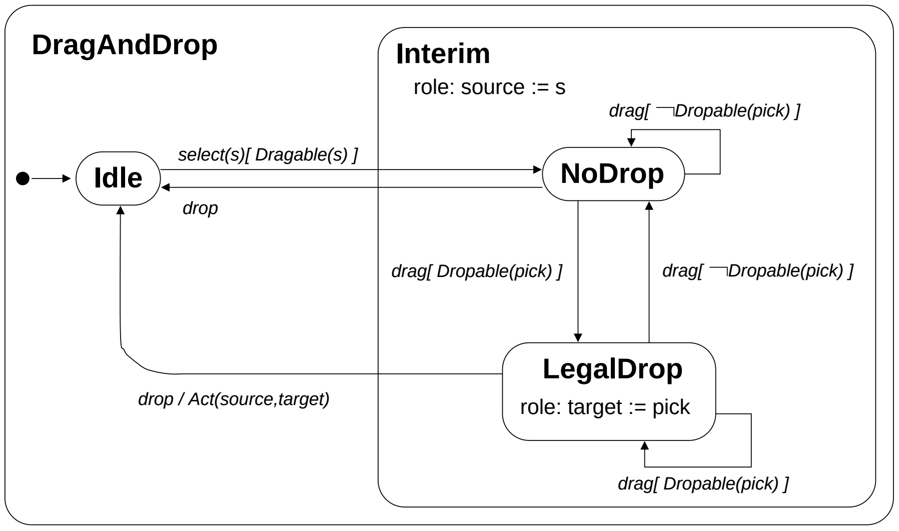


Figure 3. Specification schema for the DragAndDrop gesture.

use it in the extended example (Section 4) and because it demonstrates some of the novel features of mode components. Interaction is triggered by a select event, whose parameter is the interactor that was selected for dragging. The transition is constrained by the condition Dragable, which must be a boolean function of s. If, for example, this gesture applied only to red checkers, as we will see in the extended example in Section 4, then Dragable might be the condition: $s.color = red$.

The Interim state is used to encode an event-delimited, status context. Interim is an hierarchical state with two substates—LegalDrop and NoDrop—and with a role source. Upon entry into this state, source is assigned to remember the selected interactor. When in substate LegalDrop, a drop event determines a command to be performed. Conversely, when in substate NoDrop, a drop represents an abort. Transitions between these substates are governed by the condition Dropable, which (like Dragable) must be supplied by the designer. Dropable is any condition in which the variables source and pick appear free. By convention, each occurrence of the drag event assigns the variable pick with the "top-most" object under the mouse. Picking is a service provided by UI toolkits. Because graphical objects tend to be stacked on top of one another, sometimes the object to pick is not clear. We do not address the pick-ambiguity problem in this paper.

An idiomatic use of roles and status conditions is particularly useful for implementing dragging feedback. The idiom is to use roles in one component to encode the event-delimited–status required to form status conditions in a higher level component. An informal specification of such a condition is, "User is performing a gesture in which I [the interactor] am the *drop-target*." When this condition is true, the presentation of the interactor will be visually offset. We can express the condition formally as: [self = DragAndDrop.target].

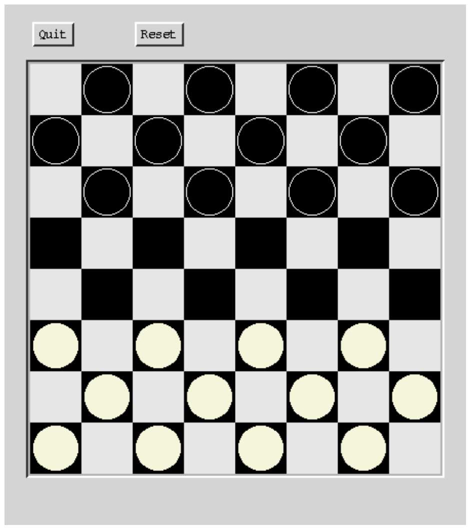


Figure 4. Interactive checkers program.

4. Example

We used MCs to construct a layered design of a larger application, the graphical checkers program that is part of the Amulet [13] source distribution (Figure 4). Players move and jump by directly manipulating checker icons. As a checker is being dragged, the system provides visual feedback to signify the legality of a drop over a particular square. This legality is a function of the player's turn and domain-object conditions, such as whether or not a target square already contains another checker.

The full specification is too large to include here, but we show the MCs for the commands and two of the four gestures. When reading these specifications, the reader should observe how well the syntactic concerns are captured in the gestures layer, while the semantic concerns are captured in the commands layer. Specifically, the command-level MC accurately captures the complex two-player dialog and some of the functional pre-conditions. For sake of brevity, we do not include all of the functional pre-conditions, but we clearly indicate any deviations in the text.

4.1. Commands: Game rules

Figure 5 shows a model of the commands that checker players perform. We model the commands—Move and Jump—as events M and J and distinguish opponents by the subscripts *Red* and *Black*. The states in this model enforce an ordering over these commands (events).

The dialog begins in state B, from which the only allowable commands are moving a black checker to an adjacent square and using a black checker to jump over a red checker. M_{Black} represents the move command; J_{Black} represents the jump command. If the user performs the move command—by dragging a black checker to an unoccupied adjacent checker square and dropping the checker in that square—a gesture will request the M_{Black} service, which will cause the MC to move into the R state. The R state is analogous to the B state except that the commands involve red checkers.

B and R are sub-states of the hierarchical states Black Turn and Red Turn respectively. These represent the contexts in which any black (respectively red) checker can be used to perform a command. The sub-states Black Jumped and Red Jumped represent a special case in which a black (respectively red) checker has just jumped over an opponent checker and is in a position to make yet another jump. Upon entry into these states, we establish the role r to remember the checker that was used in the jump. The do activities are then invoked: The checker that was jumped is removed from the appropriate set, and then the jumping checker p is physically moved to the new square sq.

The rules of checkers permit multiple successive jumps by the same checker. So from the Black Jumped state, the black opponent may jump again if she uses the same checker (r). However, because Black Jumped is a substate of Red Turn, it is also legal for any red checker to move or jump.

The final states Black Winner and Red Winner can only be entered after a jump. The condition $[blackSet = \varnothing]$, which guards the transition into Red Winner, refers to the domain object *blackSet*, which we assume is specified in the object model.

This one MC constitutes the entire Commands layer in this application. Some of the services have complex pre-conditions. For example, the pre-condition for the service $M_{Red}(p, sq)$ says "square sq is adjacent to the square that contains checker p, and sq is in a forward direction from p." For sake of brevity, we do not show these constraints in this diagram, but they are just guards on the reception of the various move and jump events.

4.2. Gestures: Checker manipulation

This layer invokes the command-services M_{Black}, M_{Red}, J_{Black}, and J_{Red}. Each service is handled by a separate instance of the DragAndDrop MC from Section 3.3. We illustrate two of the four instances: one invokes the service M_{Red} while the other invokes J_{Red}. Consider first the component that performs send $\mathsf{M}_{\mathsf{Red}}(\mathsf{source}, \mathsf{target})$ after observing a drop. We will first explain the Dropable condition. At this point in the interaction, the variable source names the object being dragged (a checker), and the variable pick names the object under the mouse, which will be a checker square[2]. The condition is:

$$\text{CheckerDialog } \textbf{enables } M_{Red}(source, pick)$$

Observe that by using the **enables** condition, the complex conditions that decide when a move is legal flow transparently from the command layer in which the information lives into the gesture. The condition for Dragable is slightly more complex because, at this point in the interaction, we do not yet have a drop object. For this we use the quantification feature:

$$\exists\, target : CheckerSquare \bullet \\ \text{CheckerDialog } \textbf{enables } M_{Red}(s, target)$$

This condition is true if there exists a CheckerSquare, (which we name target), that satisfies the event-enabled condition CheckerDialog **enables** $M_{Red}(s, target)$.

Now consider the instantiation of DragAndDrop in which the derived action is send$J_{Red}(source, target)$. The conditions Dragable and Dropable are analogous to those for moving a red checker. For Dragable, we have:

$$\exists\, target : CheckerSquare \bullet \\ \text{CheckerDialog } \textbf{enables } J_{Red}(s, target)$$

and for Dropable, we have:

$$\text{CheckerDialog } \textbf{enables } J_{Red}(source, target)$$

[2]That is we are resolving the pick ambiguity at design time. The pick operation will always return a square if a square is in the stack of objects under the mouse.

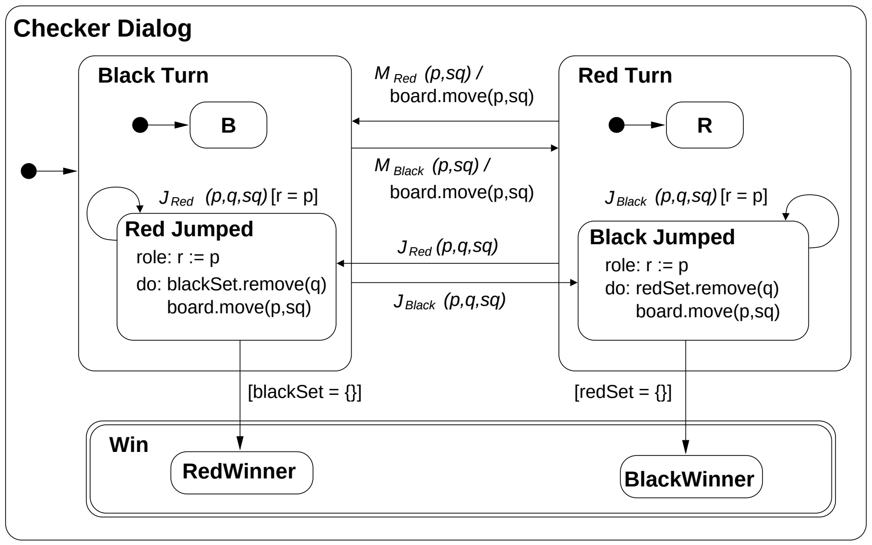


Figure 5. Dialog model for the checkers game. Events correspond to commands.

The only difference is that the event being checked is the J_{Red} event as opposed to the M_{Red} event.

These two instantiations of DragAndDrop define gestures that recognize commands from the manipulation of red checkers. A similar pair of instantiations defines the manipulation of black checkers. The conjunction of these four instantiations constitutes the direct-manipulation layer of this user interface. Observe that this layer is abstract with respect to the device events, such as mouse-click and mouse-motion; yet it describes exactly how the manipulation of checker objects implements the commands (events) that are named in the command layer. This layer issues the events required for it to be used by the dialog layer, and it constrains its behavior to conform to the command state represented in the dialog layer.

5. Discussion and conclusions

An important design principle is to encapsulate the implementation of a component behind an interface so that clients will not be affected by changes in the implementation. Unfortunately, operations that implement feedback must break this encapsulation. It is a waste of effort to encapsulate an implementation just to then go and “design” an interface that mirrors the implementation. Mode components relax the rigidity of a component interface so that designers do not have to waste effort in this way.

A related approach to UI layering is based on the use of *agents* to specify UI components [2, 15]. Agents are founded in theories of communicating processes (e.g., [10, 6]), and their interaction is defined in terms of observation and rendezvous-style synchronization rather than a transfer of sequential control. Like MCs, agents relax the rigid interface of layered components, and they provide some of the benefits of our approach. We developed support for composing components in the agent style in [22, 23]. Other researchers have come up with elaborate theories on how to use agents to implement layered components (c.f., [18, 12]). Unfortunately, these theories use features that are either difficult to implement, or that cannot fully utilize the facilities provided by a modern toolkit. We discuss these issues in greater detail in [21] and conclude that agent-based approaches to layering DMIs are unwieldy.

There is also a long history of research into the architecture of interactive systems. Good surveys of this research can be found in Olsen [16] and Bass, Clements, and Kazman [4, Ch. 6]. These generic UI architectures provide heuristic design guidance. We believe, however, that real designs often must deviate from these guidelines to handle thorny issues, such as feedback. We believe that our work helps to adapt a conceptual UI architecture into a workable implementation architecture.

This work contributes to the body of automated software engineering a new model for dealing with features that are difficult to integrate into a layered design. We envision mode components being used the framework of an application generator. Designers will construct and maintain mode components, and an application generator will generate the code from these components. Mode components add another dimension to behavioral modeling in much the same way that GenVoca components added another dimension to object modeling.

The work also raises some questions, which we did not try to address in this paper. The most interesting issues concern automation. While there are nice approaches to automate view notification, the automated generation of precondition evaluators from the quantified **enables** conditions requires more investigation. Our next step is to investigate this problem and to incorporate MCs into an environment that supports this degree of automation.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] G. D. Abowd. *Formal Aspects of Human-Computer Interaction*. PhD thesis, Oxford University, 1991.

[3] G. D. Abowd and A. J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. In *Proceedings of the ACM SIGSOFT'94 Symposium on the Foundations of Software Engineering*, New Orleans, Louisiana, December 1994.

[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. SEI Series in Software Engineering. Addison Wesley, 1998.

[5] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.*, 1(4):355–398, October 1992.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comp. Netw. ISDN Sys.*, 14(1), 1987.

[7] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1999.

[8] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[9] D. Harel. On visual formalisms. *Commun. ACM*, 31(5), 1988.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] G. E. Krasner and S. T. Pope. A cookbook for using the model view controller user interface paradigm in smalltalk. *Journal of Object Oriented Programming*, 1(3), 1988.

[12] P. Markopoulos, J. Rowson, and P. Johnson. On the composition of interactor specifications. In *Formal Aspects of the Human Computer Interface, BCS-FACS Workshop*, 1996.

[13] B. A. Myers et al. The Amulet environment: New models for effective user-interface software development. *IEEE Trans. Softw. Eng.*, 23(6), 1997.

[14] B. A. Myers, D. A. Giuse, and B. V. Zanden. Declarative programming in a prototype instance system: Object oriented programming without writing methods. In *Proceedings of OOPSLA*, 1992.

[15] L. Nigay and J. Coutaz. Building user interfaces: Organizing software agents. In *ESPRIT'91*, 1991.

[16] D. R. Olsen. *User-interface management systems: Model and algorithms*. Morgan Kaufmann, 1992.

[17] D. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2), 1979.

[18] F. Paternò. A theory of user-interaction objects. *Journal of Visual Languages and Computing*, 5:227–249, 1994.

[19] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[20] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–63, 1983.

[21] R. E. K. Stirewalt and G. D. Abowd. Practical dialogue refinement. In *Proceedings of the Fifth International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'98)*, June 1998.

[22] R. E. K. Stirewalt and S. Rugaber. Automating user-interface generation by model composition. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, 1998.

[23] R. E. K. Stirewalt and S. Rugaber. The model-composition problem in user-interface generation. *Automated Software Engineering*, 7(2), Apr. 2000. To appear.

[24] R. N. Taylor et al. Chiron-1: a software architecture for user interface development, maintenance, and run-time support. *ACM Trans. on Computer-Human Interaction*, 2(2):105–144, June 1995.

[25] H. Trætteberg. Modeling direct manipulation with referent and statecharts. In *Proceedings of the Fifth International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'98)*, June 1998.

[26] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.

[27] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[28] M. Young, R. N. Taylor, and D. B. Troup. Software environment architectures and user-interface facilities. *IEEE Trans. Softw. Eng.*, 14(6):697–708, June 1988.