ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

Using Object-Z to Compare the MVC and PAC Architectures

A. Hussey and D. Carrington

Published in Collaboration with the
British Computer Society

Using Object-Z to Compare the MVC and PAC Architectures

Andrew Hussey and David Carrington

Software Verification Research Centre

Department of Computer Science, The University of Queensland

Brisbane, Qld, 4072, Australia

email: {ahussey,davec}@cs.uq.edu.au

Abstract

Object-oriented architectures for Graphical User Interfaces (GUI's) model the interface as a composition of interacting objects which each present an internal state to the user and provide operations on that state. Two such architectures are Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC). We use Object-Z to describe examples of the MVC and PAC architectures. We compare the characteristics of the architectures and consider the efficacy of Object-Z as a tool for modelling user-interface architectures.

Keywords: Human-computer interface, software architecture, Object-Z, MVC, PAC.

1 Introduction

Interactive systems are often complex. With complexity comes the need for structure to enhance maintainability of the system. The structure of an interactive system is provided by its architecture. Architectures split a system into components and connectors. Re-usable architectures give designers a “head-start” in development by helping them avoid design errors. Architectures and methods for the implementation of interactive systems have been the focus of much recent research [12]. In this paper we demonstrate that formal models can assist in explaining user-interface architectures.

Most architectures for interactive systems divide interactive system implementation into three domains: input/output definition, application and dialogue sequence which correspond to the lexical, semantic and syntactic levels in a linguistic interaction [20]. The system can be further structured as a network of interactors each dealing with different subsets of the human-computer dialogue. An interactor is an interface component which mediates between the underlying application and the user [8]. Interactors accept stimuli, undergo state transformations and respond by sending stimuli to other interactors [1]. Interactors have a presentation aspect which reflects the internal state of the application [7]. Interactors are an example of the “observer” pattern identified by Johnson [13]. In the case of interactors, the observer pattern directs a separation of application data from dialogue (syntactic and lexical) concerns by nominating a subject (the application state) and an observer (the interface).

Object-oriented architectures view an interactive system as composed of interacting *objects*. Interactors are objects because interactors are real-world entities and are defined by their state and operations upon that state. A description of a GUI, based on interactors, parallels the presentation of the interface. So, an interactor-based description is frequently easier to understand than a non-interactor description. The distribution of information inherent in object-oriented approaches better matches the cognitive organisation of human knowledge [6]. In addition, such distribution allows independent modification of interactors, distribution of the application and multiple dialogue threads.

Two architectures which adopt an interactor-based view are Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC). Both architectures apply an object-oriented approach to the decomposition of an interactor into constituent components. To assist in describing the MVC and PAC architectures, we present a case study of a clock interactor for each architecture, using the Object-Z language ¹. Object-Z [9] is an object-oriented extension of the Z notation [22] (see Appendix A). The clock consists of synchronised analogue and digital components. The design has formed the basis for implementations using Smalltalk and Tcl/Tk. In Section 2 we specify the classes required for

¹The designs comprising the case study have been checked using the *wizard* type checker developed by the Software Verification Research Centre at the University of Queensland.

modelling time. Section 3 provides a model of a clock interactor using the MVC architecture and Section 5 provides a model of the same interactor using the PAC architecture.

The descriptions are designs, rather than specifications. Furthermore, because we abstract from presentation concerns (i.e., how input is obtained and how output is produced) the designs are not complete. We abstract from these concerns because they are not a part of, or relevant to appreciating the differences between, the MVC and PAC architectures. It is arguable that from a constructor's perspective, the parts of the design which we do not provide are the very parts which it is most necessary to describe. However, construction of the case-study is not the focus of this paper. Rather, our purpose is to compare the MVC and PAC architectures by providing simple models of the same system, using the same notation, for each architecture. Our aim in applying formal methods is to take advantage of the increased clarity that formal methods produce. By providing formal models of the MVC and PAC architectures for a simple example system, we hope to illuminate the differences between the two architectures. In [11], we provide a model of the MVC and PAC architectures which is not case-study-based. In this paper, we interpret the literature on the MVC and PAC architectures where they are ambiguous. Where the need for such an interpretation arises, we identify it, the alternatives which are available and the choice which we make.

2 Modelling Time

A clock deals with times, so we begin by defining required schemas and types.

$$\left| \begin{array}{l} LoHr, LoMin, HiHr, HiMin : \mathbb{N} \\ LoHr = 0 \wedge HiHr = 23 \\ LoMin = 0 \wedge HiMin = 59 \end{array} \right. \quad \begin{array}{l} Minute == LoMin \dots HiMin \\ Hour == LoHr \dots HiHr \end{array}$$

On an analogue clock, time may be in either AM or PM mode.

$$ModeType ::= AM \mid PM$$

The function *switch* allows us to toggle between modes.

$$\left| \begin{array}{l} switch : ModeType \rightarrow ModeType \\ switch(AM) = PM \\ switch(PM) = AM \end{array} \right.$$

We define constants to restrict the allowable hour settings for an analogue clock.

$$\left| \begin{array}{l} AnalogueHiHr, AnalogueLoHr : \mathbb{N} \\ AnalogueHiHr = 12 \\ AnalogueLoHr = 1 \end{array} \right.$$

A *Time* object may accept input from the user and tick.

<i>Time</i>	
$\begin{array}{l} min : Minute \\ hr : Hour \end{array}$	
$\begin{array}{l} \textit{Update} \\ \hline \Delta(min, hr) \\ hr? : Hour \\ min? : Minute \\ \hline min' = min? \\ hr' = hr? \end{array}$	$\begin{array}{l} \textit{Tick} \\ \hline \Delta(min, hr) \\ \hline min' = (min + 1) \bmod (HiMin + 1) \\ min' = LoMin \Rightarrow hr' = (hr + 1) \\ \quad \bmod(HiHr + 1) \\ min' \neq LoMin \Rightarrow hr' = hr \end{array}$

We define an *AnalogueTime* as a sub-class of *Time* in which *Tick* is redefined to restrict times to analogue values and updates are permitted for analogue inputs only.

$\text{AnalogueTime} \text{ } \underline{\hspace{10em}}$ $\text{Time redef}[Tick]$	
$\text{mode} : \text{ModeType}$ $hr \leq \text{AnalogueHiHr}$ $hr \geq \text{AnalogueLoHr}$	INIT $\text{Set mode to correspond to current system time}$
Update $hr? \leq \text{AnalogueHiHr}$ $hr? \geq \text{AnalogueLoHr}$	
Tick $\Delta(min, hr, mode)$ $min' = (min + 1) \bmod (HiMin + 1)$ $min' = LoMin \Rightarrow hr' = (hr \bmod \text{AnalogueHiHr}) + 1$ $min' \neq LoMin \Rightarrow hr' = hr$ $(hr' = \text{AnalogueHiHr} \wedge min' = LoMin) \Rightarrow mode' = \text{switch}(mode)$ $(hr' \neq \text{AnalogueHiHr} \vee min' \neq LoMin) \Rightarrow mode' = mode$	

3 MVC

The MVC architecture divides an interactor into three components: model, view and controller. In the MVC architecture, the model component is the abstraction of the interface object, the view depicts the model (i.e., the user's perception of the abstraction) and the controller deals with input associated with that interface object [14]. Each of the model, view and controller components are objects.

Our description of the MVC architecture is abstract, i.e., it is not a description of the MVC framework provided by Smalltalk. Hence, we make explicit the existence of interactor objects, although these are *not* part of the Smalltalk framework. Likewise, the operations which are available on objects in our description of MVC are not intended to describe precisely the same message passing as occurs in Smalltalk.

3.1 Simple Clock

We define a simple clock interactor, consisting of a model-view-controller triple. The notation used is described in detail in Appendix A. The view and controller are unique to the interactor (hence they are contained, $\odot$), but the model need not be. We use the polymorphic operator ($\downarrow$) to indicate that a clock interactor is any object whose model, view and controller are instances of classes inheriting from *ClockModel*, *ClockView* and *ClockController* respectively. The only operations permissible upon a *ClockInteractor* (as defined by the export list, prefixed $\downarrow$) are *AlterRefresh* and *Tick*.

$\text{ClockInteractor} \text{ } \underline{\hspace{10em}}$ $\downarrow(\text{AlterRefresh}, \text{Tick})$	
$m : \downarrow \text{ClockModel}$ $v : \downarrow \text{ClockView} \odot$ $c : \downarrow \text{ClockController} \odot$	
$v.model = m \wedge v \in m.dependents$ $c \in v.controller \wedge c \in m.dependents$ $c.model = m \wedge c.view = v$	

$$\begin{aligned}
 \text{Alter} &\hat{=} c.\text{Update} \parallel m.\text{Update} \\
 \text{Refresh} &\hat{=} m.\text{Refresh} \parallel (\wedge d : m.\text{dependents} \bullet d.\text{Refresh}) \\
 \text{AlterRefresh} &\hat{=} \text{Alter} \circ \text{Refresh} \\
 \text{Tick} &\hat{=} m.\text{Tick} \parallel (\wedge d : m.\text{dependents} \bullet d.\text{Refresh})
 \end{aligned}$$

The sequential relationship between alteration of the clock interactor (i.e., acceptance of input) and refreshing of all associated dependents, is modelled using sequential composition ($\circ$). The new time is communicated from the model to the dependents using the concurrency operator ($\parallel$). A clock may tick without any user input. Informally we assert that this occurs once every minute. When the clock ticks, all of the dependents are refreshed. We describe the update of the set of dependents using distributed conjunction ($\wedge$). The effect of *Tick* is to unify the output *vtime!* from *m.Tick* with the input *vtime?* for each *Refresh* operation on a dependent (in this case they are all *ClockViews*). The effect is to update the time displayed by each view.

A model is an object “which implements some application-domain-specific functionality and/or state” [21]. The interface of a model includes methods that allow access to those aspects of the state which are rendered visible to the user through the view.

A clock model is only aware of the current hour and minute. The attribute *storedTime* in a *ClockModel* is contained because no two clock models alter the same time attribute. A tick operation increases the time once every minute. A model may have dependents. Alterations to a model are propagated to all associated dependents. Note that for conciseness we do not define the abstract class *Interactor* in this paper; it is defined in [11].

A model broadcasts a message to all its dependents when it changes. The message may be parameterised by an aspect. Dependent views which display the altered aspect may query the model and update accordingly [21]. For our clock example however, we simply update all the dependents.

A view has an associated model and may have an associated controller. The view for a clock displays a representation of a time.

Since views correspond to windows, window nesting leads to the notion of sub-view and super-view [21]. Hence views may be organised in a hierarchy [17, p.196]. The appearance of a system is structured by organising views within objects whose behaviour is analogous to that of a picture frame [19, p.238]. These structural objects are not strictly part of the MVC architecture and are merely a mechanism for ensuring on-screen proximity of related views. We also do not deal with composite views (which are an application of the “composite” pattern) because we do not consider composite views to be central to the MVC architecture (although they are a part of implementation frameworks based on MVC such as VisualWorks).

A controller has an associated model and view. A model may be updated (via the controller), effecting a change to the current hour and minute. Controllers may create child interactors such as dialogue boxes. We don’t need to describe the details of child interactors. An update occurs only if the associated restrictions on valid input for times are satisfied. The description presented is abstract hence the new time in an *Update* is supplied directly by the environment.

Alterations to the model which affect the controller may be passed to the controller by the associated view [15, p.96] or the controller may be registered as a dependent of the model. In our description, we adopt the latter approach. In addition, the controller may access the view because this allows alteration of the view where the model is not changed (e.g. a change to the way in which the view depicts the model such as an alteration to the display resolution).

The primary function of the controller is to manage mouse/keyboard event bindings (e.g., for pop-up menus). If an input event (mouse, keyboard etc.) requires modification of application-specific data, the controller notifies the model accordingly [21]. Sub-interactors of a controller are view-controller pairs whose model may be the application model. Figure 1 shows an MVC interactor whose controller creates a sub-interactor. Such view-controller pairs

Figure 1: An example MVC interactor

include pop-up menus and dialogue boxes [14]. Note however that although we allow for this possibility, the clock controllers introduced in this case study do not have any child interactors. In many descriptions of MVC, the creation of sub-interactors is ambiguous. The above description is our interpretation of how such creation can be reconciled within the MVC architecture.

The model usually is responsible for creating its own views and controllers when it is initialised. Therefore the view and controller could be depicted as child objects of the model. However because the model need not always create the controller and view, we follow the established convention (e.g., [19]) for depicting the MVC architecture.

The primary system model may also create “satellite” interactors, such as buttons, which have their own models but which invoke methods of the primary model. In the case of a *ClockModel*, there are no such satellite interactors.

Alterations to the model of one interactor may affect other interactors. The relationship between interactors may be expressed as a mathematical relationship between models. Related interactors exchange messages through message passing between their models. Alternatively, relationships between interactors may be implicit from a shared model [19, p.205]. This obviates the need for message passing and so is usually the better approach.

3.2 Composite Clock

To illustrate model sharing, we develop a description of a *CompositeClock* consisting of an analogue clock and a digital clock. The interface appears in Figure 2.

Figure 2: The composite clock interface

The structure of our design is illustrated diagrammatically in Figure 3.

Figure 3: The MVC inheritance hierarchy for the composite clock

The notation used is BON (Business Object Notation) developed by Nerson [18]. The double width arrows indicate a client relationship between classes. Only the principal client relationships are indicated, though others exist (e.g., an instance of *AnalogueController* is an attribute of *AnalogueClock*).

We define a composite clock class. Note that this class would probably not appear in an implementation. Instead, the *CompositeClockModel* would create the views and controllers for the analogue and digital clocks as part of its

own initialisation.

<i>CompositeClock</i>
$analogue : AnalogueClock$ $digital : DigitalClock$
$analogue.m = digital.m$
$AnalogueAlterRefresh \hat{=} analogue.AlterRefresh$ $DigitalAlterRefresh \hat{=} digital.AlterRefresh$ $Tick \hat{=} (analogue.Tick \wedge digital.Tick)$

In a composite clock, several views share the one model, so the model for the analogue and digital clock's is a *CompositeClockModel*. We exploit model sharing as shown in Figure 4.

Figure 4: Structure of our MVC clock

A *CompositeClockModel* provides operations which accept input from, and update, analogue and digital displays.

<i>CompositeClockModel</i>
<i>ClockModel</i> [<i>DigitalUpdate/Update</i>]
<i>AnalogueUpdate</i>
$\Delta(storedTime)$ $mtime? : Time$
$storedTime'.min = mtime?.min$ $storedTime.hr > AnalogueHiHr \Rightarrow$ $storedTime'.hr = (mtime?.hr \bmod AnalogueHiHr) + AnalogueHiHr$ $storedTime.hr \leq AnalogueHiHr \Rightarrow storedTime'.hr = mtime?.hr$

3.3 Analogue Clock

We define an analogue clock interactor using the MVC architecture.

<i>AnalogueClock</i>
$\uparrow(AlterRefresh, Tick)$ <i>ClockInteractor</i> redef [<i>Alter</i>]
$m \in \downarrow CompositeClockModel$ $v \in \downarrow AnalogueView$ $c \in \downarrow AnalogueController$
$Alter \hat{=} c.Update \parallel m.AnalogueUpdate$

The view displayed for such an interactor represents an analogue time.

An analogue clock controller only accepts analogue times as input because an analogue display only represents a 12 hour range. The analogue clock uses direct-manipulation to input times, hence the clock's controller does not create any sub-interactors.

We do not describe the digital clock in this paper. Its structure (in terms of MVC) is similar to the analogue clock.

4 PAC

The PAC architecture divides an interactor into three components: presentation, abstraction and control. In the PAC architecture, the presentation component defines an interactor's appearance and the input lexemes it accepts, while the abstraction defines the minimal state representation; dialogue sequence (syntax) may be controlled by either the presentation or the abstraction. Communication between the abstraction and presentation components occurs via the control component. Message passing between interactors occurs between control components in each interactor.

4.1 Simple clock

We introduce a *HierarchicalClockInteractor* class to model an interactor in a PAC hierarchy whose behaviour can be characterised as that of a clock and which is able to form part of a larger composite clock interactor (combining multiple clock interactors).

An interactor may accept input which alters the interactor's state. Any of the children of an interactor may be responsible for an input. The use of a shared state ensures that updates to the presentation component (p) are implicitly communicated to the control component (c) and abstraction component (a). We discuss this in more detail below. Inputs are propagated to parent interactors. A clock interactor may *Tick* autonomously.

The presentation of a PAC object is partly determined by the presentation of component objects [6]. There is some ambiguity between [4], [5] and [6] as to whether child interactors are created by the parent presentation. We choose to regard the presentation as composed (in part) of sub-interactors because this simplifies the creation of interactors. The clock's presentation may be altered in an *Update* operation. This alteration is reflected by a corresponding change to the abstract state.

The clock abstraction performs tick operations which update the time. The attribute *storedTime* is contained because the abstraction actually stores the time, whereas the control and presentation components reference the stored time.

Figure 5: An example PAC interactor

The control component maintains knowledge about the presentation and abstraction components. We regard such contextual information as including formulae relating the abstraction to enclosing objects and auxiliary information such as dialogue history and help [6]. Again, the literature is ambiguous as to whether the conversion between states occurs in the child or parent. We choose to regard conversion as occurring in the child because this approach is simpler. The control represents the boundary between interface (lexical) and application (semantic) concerns. Control components communicate with their respective parent interactors.

The PAC structure shown in Figure 5 corresponds to the composite clock depicted in Figure 2. Sub-interactor 1 represents the analogue display and sub-interactor 2 represents the digital display. The composite control (C) ensures synchronisation of the analogue control (C_1) and the digital control (C_2).

We provide the following example based on [6], describing the events following manipulation of the digital clock to enter a new time. When P_2 is acted upon to alter the digital presentation, a message to this effect is sent to C_2 . C_2 both updates the state A_2 and notifies C of the update. C translates the update from P_2 to the correct form for A (i.e., 24 hour time - in this case no change is required). C 's task is to transmit the appropriate update to C_1 so that the analogue representation, P_1 , is updated. C_1 "knows" the formula relating the state A_1 to A . This formula converts the 24 hour time recorded by A to the 12 hour time recorded by A_1 .

An *HierarchicalClockControl* object can communicate with other such control components in an interactor hierarchy. The *UpLevel* operation instructs a parent interactor to receive an input from a child interactor, broadcast it to all children and propagate the input up the interactor hierarchy if possible. We use angelic choice ($\parallel$) to specify that either there is a parent, so propagation will be possible, or there is not (in which case no action is taken). The *Broadcast* operation instructs a child to receive a time from the parent (instructing) interactor and broadcast it to all its children.

4.2 Composite Clock

In Section 3, we developed a description of an MVC architecture for a composite clock interface. For comparative purposes, we repeat this exercise for the PAC architecture. The structure of this design is illustrated diagrammatically in Figure 6. As for MVC, only the principal client relationships are indicated, though others exist.

A digital clock is similar to an analogue clock except that time is represented using a 24 hour format. The digital and analogue clocks can be combined as sub-interactors of a composite clock presentation. The control of such a composite clock is, again, a *HierarchicalClockControl*. For brevity we do not detail the digital or composite clock classes here. Further details can be found in [11].

Figure 6: The PAC inheritance hierarchy for the composite clock

4.3 Analogue Clock

We define an analogue clock interactor using the PAC architecture. We use the analogue time class within the *AnalogueAbstraction* class because PAC does not use abstract state sharing (as does the MVC example). In the MVC description, the analogue display was simply a view on the digital abstract state. For PAC, the analogue display depicts its *own* abstract state.

The components of an analogue clock interactor may be defined.

An analogue clock only represents time on a 12 hour dial.

An *AnalogueControl* performs the same functions as a standard *HierarchicalClockControl* except that updates received from the parent control require conversion to an analogue format (we assume the parent uses digital format).

ReceiveParentTime

$$\begin{aligned}
&time'.min = newtime?.min \\
&newtime?.hr \neq LoHr \wedge newtime?.hr \neq AnalogueHiHr \\
&\quad \Rightarrow time'.hr = newtime?.hr \bmod AnalogueHiHr \\
&newtime?.hr = LoHr \vee newtime?.hr = AnalogueHiHr \\
&\quad \Rightarrow time'.hr = AnalogueHiHr \\
&(newtime?.hr \geq AnalogueHiHr) \Rightarrow (time'.mode = PM) \\
&(newtime?.hr < AnalogueHiHr) \Rightarrow (time'.mode = AM)
\end{aligned}$$

5 Conclusions

Object-Z allows the succinct expression of several key aspects of the MVC and PAC architectures. We use Object-Z to model interactors in the context of a case study. The class construct in Object-Z provides a suitable structural mechanism for modelling interactors and their components.

Object-Z clarifies message passing between components in each architecture. For example, in the case of the PAC architecture, an update to the presentation (i.e., input from the user) is propagated to the abstraction (via the control), to the parent interactor in the PAC hierarchy and to all the child interactors. This is achieved by the operation *AlterRefresh*. Likewise, in the case of the MVC architecture, an alteration to the model (via the controller) must be propagated to the dependents of that model (which include the view). We ensure this by offering only the composite operation *AlterRefresh* rather than the operations *Alter* and *Refresh*.

The containment construct captures the notion that an attribute may denote a unique object reference (i.e., the attribute is said to be “contained”). Containment is useful to model the relationship between an interactor and its components. So for example, the *Presentation*, *Abstraction* and *Control* components of a *HierarchicalClockInteractor* are contained.

Object-Z captures the *sequence* of messages that occurs after a user action. The sequential composition and concurrency operators permit modelling of temporal relationships between operations. Concurrency may be used to model information passing between objects. The sequence of messages in an implementation may be deduced from the direction of information flow. For example in the MVC architecture, the controller accepts input from the user; the operation *Alter* accepts input, via the *ClockController* and passes that input to the *ClockModel*. Sequential composition may be used when an explicit temporal ordering is required but information passing is not involved. For example, in the PAC architecture, the temporal relationship between an *Update* by the *ClockPresentation* and an *UpLevel* and *BroadCast* by the *ClockController*, is captured by sequential composition. Note that where the order of performance is irrelevant, we use conjunction ($\wedge$).

An Object-Z description provides a simpler model of system structure than corresponding implementations using Smalltalk (for MVC) and Tcl/Tk (for PAC). The Object-Z model is simpler because we abstract from presentation concerns (i.e., how input is obtained and how output is produced). This abstraction allows us to emphasise the structures and message passing which instantiate the components and connectors for each architecture.

The description of the PAC architecture is unusual in that the objects which provide the system’s external interface are not those higher in the control hierarchy but the sub-interactors at the leaf level. In most Object-Z descriptions, system operations are defined as acting on a *System* class (or equivalent) which contains instances of the various system objects. However in our case, the operations on sub-interactors invoke corresponding operations on parents. It is possible to add a *System* class to the description of the PAC architecture, however we believe that by not doing so, we model the true system structure more closely.

From the formal descriptions in the preceding section and our understanding of the architectures, we note that the primary differences between the PAC and MVC architectures are:

- how synchronisation of related interactors is achieved, and
- the location of input and output responsibilities.

The PAC control component has no direct correspondent in the MVC architecture [2, p.171]. Components in the MVC architecture maintain their consistency with other interactors through message passing, however, unlike PAC, the MVC architecture does not dictate the relationship between interactors. Alternatively a model may be shared between

several views and controllers. The clock example described in Section 3 uses the latter approach. So a composite clock consists of an instance of *CompositeClockModel* which is shared by analogue and digital view-controller pairs. Altering the model via either controller updates *both* views (and all other dependents). Because the MVC architecture is not hierarchical, the control relationship is specific to each system. For complex systems, the structure introduced by the PAC architecture provides easier maintenance than the MVC architecture. For simple systems, the approach adopted in the MVC architecture is sufficient. In simple systems, the control component is usually just a message relay between the levels in the PAC hierarchy and the presentation and abstraction components of each interactor. In that situation, the additional message passing required is wasteful. For systems of intermediate complexity, the PAC hierarchy provides structure but the control components may have few roles beyond connecting interactors. In that case, merging the control and abstraction components may help reduce message passing. Such an approach would be similar to the architecture proposed by Cockton [3]. For complex systems, the control component usually has several roles and in that case the full PAC architecture may be justified. The PAC architecture does not acknowledge the possibility of shared abstractions, however there is no reason to disallow such sharing. In our Object-Z description, we capture the uniqueness of abstractions by asserting that they are contained (as indicated by the symbol ©) within the corresponding PAC *ClockInteractor*. In contrast, the model components of MVC *ClockInteractors* are not contained. Shared abstractions would constitute a useful extension to the PAC architecture, providing additional flexibility. Although higher level PAC interactors abstract from lower level interactors, the control hierarchy and the lack of shared abstractions produces some duplication of state information between abstractions.

Output behaviour in the MVC architecture is handled by the model component sending appropriate messages to the view and input behaviour is provided by the controller. In PAC, input and output behaviour is combined in the presentation component. We model this distinction in Object-Z by providing both *Update* and *Refresh* operations for the *Presentation* component of the PAC architecture. The *Update* operation in the MVC architecture is provided by the *Controller* and the *Refresh* operation by the *View* (corresponding operations in the *Model* accept input and produce data for output). The MVC architecture regards output as relating primarily to the depiction of abstract system states whereas the PAC architecture regards output as encompassing both the creation of sub-objects involved in human-computer dialogues (pop-up menus, entry widgets etc.) and the depiction of abstract system states. We model this in Object-Z by allowing the *Control* to have child *Interactor* objects. Thus, the presentation component is the only part of a PAC interactor with which users interact. In the MVC architecture, users interact with both the view and control components [2, p.173].

Separation of input and output may be undesirable because such separation results in increased message passing for direct manipulation systems [16]. For such systems, true separation of input and output may be difficult to achieve because input events are often “strongly related to immediate feedback” (i.e. without state change) [5]. Also such separation is not supported by many toolkit approaches such as Motif. However decoupling the controller and view enhances flexibility and reuse [10, p.4]. So for example, the implementation of *AnalogueView* in our Smalltalk implementation of the composite clock, could be re-used in building a clock which told the time for several different time zones (e.g., both Sheffield and Brisbane). Changes may be made to either input or output syntax independently. By encapsulating the response mechanism (i.e., to input) in a controller object, the MVC architecture makes it simple to substitute one controller for another and so alter the way in which a view responds to input.

In many cases, the choice of architecture is pragmatic [13]. A project which is to be implemented using Smalltalk will use MVC because Smalltalk provides many of the necessary objects (e.g., Model-View-Controller triplets for commonly used interface constructs). Likewise, a developer who expects to implement using a toolkit such as Motif will probably choose PAC because of the difficulty of separating input and output concerns. Being aware of the architectural implications of particular frameworks (Smalltalk or Motif) may influence the developer’s choice of framework.

6 Acknowledgments

The authors wish to thank Jin Song Dong for helpful discussions on containment and polymorphism and Graeme Smith for commenting on a draft of this paper. We also thank Wendy Johnston for helping type-check the Object-Z descriptions in this paper.

Appendix A: Modelling using Object-Z

In an Object-Z description, a class is defined by a named box encapsulating an export list, state, initialisation and operations. The export list (prefixed by $\downarrow$), defines which operations are visible to users of the class. If there is no export list, then implicitly all operations are visible. The state schema is un-named and contains attribute declarations and a constraining invariant. Some attributes may be dependent on other attributes (i.e., their value is always derivable from those other attributes). Such dependencies are denoted by a Δ on a separate line preceding the dependent attributes (e.g., top). The nature of this dependency is captured by the class invariant. The initialisation schema is labelled *INIT* and defines the initial state of instances of the class. An operation schema is divided into two parts. The upper part defines the context of the operation including inputs and outputs. The Δ -list defines which state attributes are altered by an operation. State variables which are not listed in the delta-list are unchanged by an operation. The lower part defines a predicate relating the initial and final states of the operation. A simple example class defining a stack of natural numbers appears below:

Classes may inherit from other classes. Inheritance results in a merger of state, *INIT* and operation schemas which have the same name. For example, we define a new stack class which can return its size:

A class may be instantiated. For example, the declaration $stack : Stack$ defines an attribute *stack* of type *Stack*. The polymorphic operator ($\downarrow$) defines the type of an attribute as any class which inherits from the specified base class. Operations applied to such an attribute must be polymorphic, i.e., regardless of the actual class of the object, the operation must be able to proceed. For example, the declaration $stack : \downarrow Stack$ defines an attribute *stack* which can be an instance of any class which inherits from *Stack*. The containment operator ($\odot$) defines an ownership relationship between class instances. For example, the declaration $stacks : \mathbb{P} \downarrow Stack \odot$ ensures that the instances of type $\downarrow Stack$ which define the set *stacks* can be contained by no other object instance but can be referenced.

We can define a set of stacks as follows:

$$\begin{aligned}
Transfer &\triangleq [s1?, s2? : stacks] \bullet s1?.Pop \parallel s2?.Push \\
Push &\triangleq [s? : stacks] \bullet s?.Push \\
Pop &\triangleq [s? : stacks] \bullet s?.Pop \\
TwoPops &\triangleq [s1?, s2? : stacks] \bullet s1?.Pop \circ s2?.Pop \\
PopAll &\triangleq \wedge s : stacks \bullet s.Pop \\
PopEither &\triangleq [s1?, s2? : stacks] \bullet s1?.Pop \sqcup s2?.Pop \\
Top &\triangleq [s? : stacks] \bullet s?.Top \\
Size &\triangleq [s? : NewStack \mid s? \in stacks] \bullet s?.Size
\end{aligned}$$

Operations and attributes of class instances may be accessed. For example, the term $s.elements$ denotes the attribute $elements$ of s while the term $s.Push$ denotes the operation $Push$ applied to s . Operations may be combined using a calculus which extends the Z schema calculus [22], including conjunction ($\wedge$) and sequential composition ($\circ$). Object-Z provides two further operators:

- concurrency ($\parallel$) which conjoins operation schemas (i.e., merges state and predicates) and identifies inputs with outputs. We illustrate concurrency in the operation $Transfer$, which selects two stacks, pops one stack and pushes the result on the other.
- angelic choice ($\sqcup$) which selects one of two operations according to which can be performed. If both operations can be performed, a non-deterministic choice is made between the two. We illustrate angelic choice in the operation $PopEither$ which pops either of two selected stacks.

Operators may be applied over a set of instances, e.g., the operation $PopAll$ specifies an operation in which all instances in the set $stacks$ are subject to the operation Pop . The meaning is the same as $s_1.Pop \wedge s_2.Pop \wedge \dots$ for all $s_1, s_2, \dots$ in the set $stacks$.

References

- [1] Abowd, G. D. Agents: Communicating Interactive Processes. In Diaper, D., Gilmore, D., Cockton, G., and Shackel, B. (eds), Human-Computer Interaction – INTERACT '90. Elsevier Science Publishers B.V., 1990, pp 143–147
- [2] Bass, L. and Coutaz, J. Developing software for the user interface. Addison-Wesley Publishing Company, 1991
- [3] Cockton, G. The Architectural Bases of Design Re-use. In Duce, D. A., Gomes, M. R., Hopgood, F. R. A., and Lee, J. R. (eds), User Interface Management and Design: Proceedings of the Workshop on User Interface Management Systems and Environments. Springer-Verlag, 1990, pp 15–34
- [4] Coutaz, J. PAC, an Object Oriented Model for Dialog Design. In Rullinger, H. I. and Shackel, R. (eds), Human-Computer Interaction - INTERACT '87. Elsevier Science Publishers, 1987, pp 431–436
- [5] Coutaz, J. The Construction of User Interfaces and the Object Paradigm. In Bezivin, J., Hullott, J.-M., Cointe, P., and Lieberman, H. (eds), European Conference on Object-oriented Programming - ECOOP '87. Springer-Verlag, 1987, pp 128–130
- [6] Coutaz, J. Architectural Models for Interactive Software. In Cook, S. (ed), European Conference on Object-oriented Programming - ECOOP '89. Cambridge University Press, 1989, pp 382–399
- [7] Duke, D. J. and Harrison, M. D. Abstract Interaction Objects. Computer Graphics Forum Conference Issue September 1993; 12(3):25–36
- [8] Duke, D. J. and Harrison, M. D. Event model of human-computer interaction. Software Engineering Journal January 1995; pages 3–11
- [9] Duke, R., King, P., Rose, G., and Smith, G. The Object-Z specification language. In Korson, T., Vaishnavi, V., and Meyer, B. (eds), Technology of Object-Oriented Languages and Systems: TOOLS 5. Prentice-Hall, 1991, pp 465–483

- [10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994
- [11] Hussey, A. and Carrington, D. Comparing two user-interface architectures: MVC and PAC. Software Verification Research Centre 95-33, The University of Queensland, December 1995
- [12] Johnson, P., Drake, K., and Wilson, S. A Framework for Integrating UIMS and User Task Models in the Design of User Interfaces. In Duce, D. A., Gomes, M. R., Hopgood, F. R. A., and Lee, J. R. (eds), User Interface Management and Design: Proceedings of the Workshop on User Interface Management Systems and Environments. Springer-Verlag, 1990, pp 203–216
- [13] Johnson, R. E. Patterns and Frameworks. Report on Object Analysis and Design March/April 1995; 1(6):46–48
- [14] Krasner, G. E. and Pope, S. T. A cookbook for using Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-oriented Programming August/September 1988; 1(3):26–49
- [15] Lewis, S. The art and science of Smalltalk: an introduction to object-oriented programming using VisualWorks. Prentice Hall, 1995
- [16] Linton, M. A., Vlissides, J. M., and Calder, P. R. Composing User Interfaces with InterViews. IEEE Computer 1989; 22(2):8–24
- [17] Marchesi, M. Object-Oriented Programming with SmallTalk/V. Prentice Hall, 1994.
- [18] Nerson, J. Applying Object-Oriented Analysis and Design. Communications of the ACM September 1992; 35(9):63–74
- [19] ParcPlace Systems Inc. VisualWorks User's Guide. ParcPlace Systems, Inc., 1994.
- [20] Shevlin, F. and Neelamkavil, F. Designing the Next Generation of UIMS's. In Duce, D. A., Gomes, M. R., Hopgood, F. R. A., and Lee, J. R. (eds), User Interface Management and Design: Proceedings of the Workshop on User Interface Management Systems and Environments. Springer-Verlag, 1990, pp 123–133.
- [21] Skipper, M. C. Model-View Controller classes in Eiffel. In Schulthess, P. (ed), Joint Modular Languages Conference Modula, Oberon & friends Germany. Germany Universitaetsverlag Ulm GmbH, September 1994, pp 235–248
- [22] Spivey, J. M. The Z notation: a reference manual. Prentice-Hall, 2nd edition, 1992