Object Interactions as First Class Objects: From Design to Implementation

Mahesh H. Dodani, Kok Siew Gan and Lizette Velazquez
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
email: {dodani, gan, velaz}@cs.uiowa.edu
Phone: (319) 353-5134
Fax: (319) 335-3624

Abstract

Collaborations between objects make up the dynamic behavior of OO software. These collaborations among objects require careful design and implementation. Treating the interactions as responsibilities that are integrated in the participating objects, results in tight coupling between objects. Tight coupling increases complexity and reduces reusability. Object interactions need to be first class objects from design to implementation. Our research provides a unified approach to model and implement these interactions as first class objects. During analysis and design, they are modeled using DynaSpecs. During implementation, they are coded with a new language construct called Compositions. DynaSpecs and Compositions provide a consistent support for object interactions within each phase of the OO lifecycle.

1 Introduction

Current OO methodologies [Boo94, Cole94, Jac92, Rum91] model the static structure and dynamic behavioral aspects of OO systems. Static models use entity relationship diagrams to describe attributes in objects and their structural relationships. There are two extremes of modeling relationships: explicit and implicit models. Implicit models use attributes in the associated classes. Explicit models of relationships model them as first class citizens [Tan95].

The dynamic model aim to describe the complex interactions between objects at run time. Interactions among collaborating objects are typically identified in the analysis phase through scenarios (use cases) that expose the functionality of the system. Most methods use object interaction diagrams to model interactions during the design phase. These diagrams capture the objects involved in the interaction and the corresponding flow of message passing. The interactions among objects encompass the assumptions that each makes about other. To understand objects interactions we need to know not just the associated message flow and the participating objects, but also what the state changes and interaction behavior of each the participating objects.

In current methodologies all this information is distributed over the entire spectrum of models: object models, operational models, interaction graphs and state machines. Furthermore, in the design of object classes, the behavior of interactions is scattered across the collaborating classes. This distribution of information and behavior across models increases the redundancy of information, complexity of the classes, and the coupling among classes. This results in difficulties in establishing invariants and properties of the system, difficulty in ensuring proper access to methods, and difficulty in ensuring consistent update of participating objects to reflect dynamic changes.

The disadvantages of treating object interactions as integral behavior of the participating objects are as follows:

- increase complexity when additional interaction responsibilities are placed on the participating objects,
- tight coupling between participating objects,

- reuse of interactions may include unnecessary tie to other interactions irrelevant to the reuse,
- difficulty in handling many to many relations (in which of the participating object should the interaction responsibilities be placed), and
- abstracting the object interaction is difficult.

Object interactions should be supported as first class values in Object-Oriented software. They need to be first class objects from design to implementation. Our research provides a unified approach to model and implement these interactions as first class objects.

At analysis and design our research develops DynaSpecs. DynaSpecs supports the specification of objects and interactions as first class objects. DynaSpecs model the object interactions using state transitions. The state transitions represent events of message calls that mirror the dynamic behavior of object interactions. The formal semantics of the model is based on the idea of a history which captures the sequence of operations within interactions along with state changes undergone by the participating objects.

Current OO programming language allow programmers to implement object interactions as first class objects by using classes with variable references to participating objects. However, this implementation have the following disadvantages:

- the variable reference allows access to all public methods of the participating object, even methods that are not relevant to the interaction,
- there are no clear separation between participants in the class encapsulation, the participants are accessed through variables,
- in a statically typed language like C++, the participating objects must belong to the variable class type. A suitable participant from a different class type cannot be a participant safely.

We provide a new construct called Composition to implement object interactions. Compositions, as well as DynaSpecs allow programmers and systems designers to

- clearly define each participant in the interaction,
- focus on functionality of the interactions by defining the behavior and responsibility of each participant,
- provide an explicit dynamic lifetime of an object interaction,
- restrict visibility to only methods of participants relevant to the interaction,
- facilitate reuse of interactions by allowing parameterized interactions, and inheriting Compositions,
- provide the necessary context to clearly separate responsibilities of participants, an encapsulation of the entire interaction, encapsulation of each participant and limited access to participating objects, and
- allow the selection of suitable participating objects based on the methods they implement, not the class hierarchy they belong to.

The semantics of DynaSpecs and Compositions have been precisely defined using denotational semantics. They have been tested with prototype executable of the constructs written in ML. These constructs can be added to any existing OO system known to us. We have successfully implemented the constructs and incorporated them in an Object-Oriented development environment. In this paper, we will explain how to use our constructs to model and implement object interactions. We have modeled the MVC framework using DynaSpecs and implemented the new MVC design using our prototype language. The paper will present our results.

The rest of this paper is organized as follow. Section 2 explains and motivates the need for DynaSpecs and Composition through a case study. It begins by explaining the interactions in the MVC framework. It then models and implements them using DynaSpecs and Compositions. Section 3 concludes the paper.

2 Case Study

One of the most well known framework is a Graphical User Interface framework called Model, View and Controller [Kras88, Mod88]. We will identify the object interactions in a MVC framework, and model and implement them using DynaSpecs and Compositions. The new MVC design will be used to implement a calculator. Throughout this case study, we explain how to use interactors in modeling and implementing the calculator with MVC.

2.1 Model, View and Controller Framework

There exist three major object interactions in applications developed using (MVC) framework:

- Views register themselves as dependents of their model and keep themselves consistent with the state of their model (ie. with the internal state of the application).
- Controllers and models interact to translate user input to invoke system functionality.
- View-Controller interaction is needed to determine the controller in charge of user interaction.

Current approaches to model the MVC interactions, make the model, view and controller classes tightly coupled. For example the view class includes behavior that deals with updating itself to keep consistency with the model, behavior to interact with the controller and behavior to traverse the view hierarchy. The associated methods access the state of the model or interpret the state of the model, translates controller inputs, and finds a view that has the cursor. Understanding and reusing the MVC framework require dissecting each of the classes to pinpoint the behavior involved in each object interaction.

Our constructs allow explicit definition of each MVC object interaction. This provides a clear separation of each object's basic behavior and separation of interactions and their participants. The interactions described above, can be modeled by the following interactors.

- Consistence interactor to keep dependents consistent with the state of an object.
- Translate interactor to translate controller inputs into operations for performers.
- Control interactor to determine which controller gets control.

These interactors perform the functionality of the MVC framework. Control interactors within an application represent a tree structure responsible for invoking the correct controller to get user input. In particular, a top control interactor determines which of its sub-control wants control. Once this is determined, it sends "controlActivity" to the sub-control. In "controlActivity", it asks each of its translate interactor to translate any user input. The translate interactor asks its performers to perform the input operation. The performers make their dependents consistent using the consistence interactor. The figure bellow illustrates our explanation.

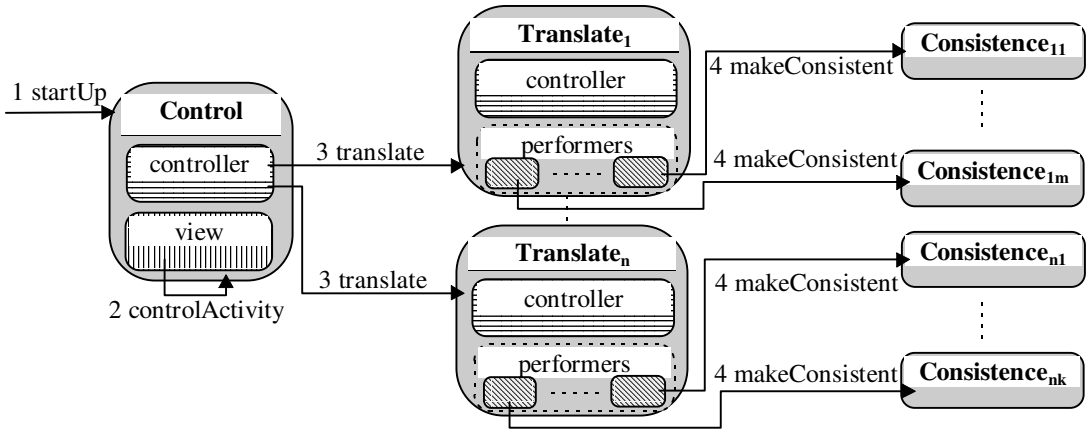


Figure 1: MVC interactors

Control interactor usually have only one corresponding translate interactor. Each translate interactor usually has one performer. Each performer participant of a translate interactor has exactly one consistence interactor.

The interactors described here can be documented using design patterns. Design patterns describe design decisions using a consistent format to help provide a common vocabulary, to help to understand and reuse the design. The format includes discussion of implementation issues and provides sample code. This format aids implementation of the design. Interactors provide a construct to implement some structural and behavioral design patterns [Gam95]. The close resemblance of the interactor code to design patterns illustrates the ease in capturing the design using interactors.

The consistence interactor keeps dependents consistent with the state of an object. According to [Gam95], the consistence interactor is similar to an observer design pattern. After a model changes its state, the view will receive an "update" message to invalidate its view. The following figure illustrates how consistence is achieved in

MVC. The figure shows the message flow between the participants in the interaction. Note that the sequence of messages is ordered as shown by the numbering.

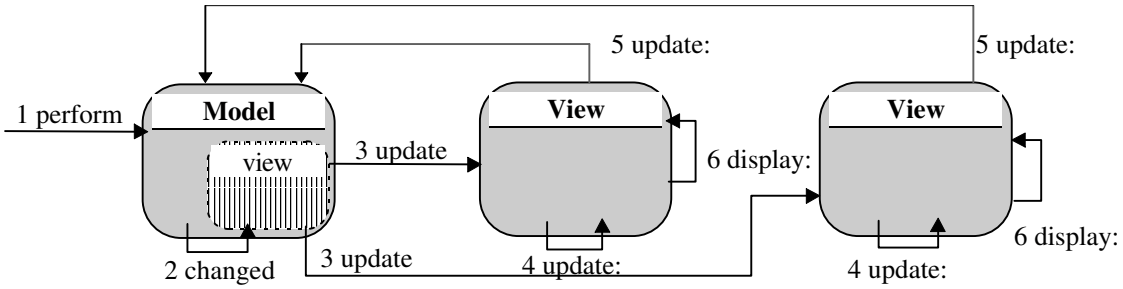


Figure 2 : Consistency in MVC

The consistence composition consists of an object participant and a list of dependent participants. It handles consistency by sending the message "makeConsistent" to the consistence composition object. The consistence object gets the object participant's value, maps it for each dependent, and passes the value to the dependents, so they may update themselves if necessary. The following diagram illustrates the message flow.

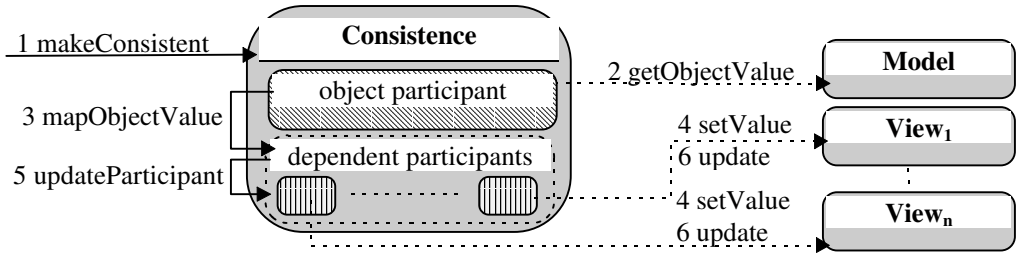


Figure 3: Consistence composition

To emphasize how the responsibilities for each object interaction are decoupled from the model and dependents, the patterns in the figure stress where they are placed. In the MVC, the Model class includes View responsibilities, the Controller includes Model and View responsibilities, and the View class includes View hierarchy responsibilities. The interactor liberates basic objects from having interaction responsibilities as part of their behavior by placing them in the corresponding interactor. Interactors act as an abstraction of the object interactions and provide communication between the collaborating objects.

2.2 Calculator Application

To reuse the MVC framework architecture, we will apply the interactors to design and implement a simple calculator. Each of the interactors described earlier will be inherited with extensions relevant to the calculator. The calculator consists of a control hierarchy which is captured through the control interactor. This hierarchy replaces the traditional view hierarchy used in current designs and implementations. Each view has a translator which translates the input received by its associated controller. When the calculator model changes, the display view must be updated. This is captured in calculator consistence interactor. Conversely, when the display view changes the number of display digits, the calculator model needs to take that into account. If the calculated result is larger than the number of display digits, the calculator model will be in an error state. This is captured in view consistence interactor. Each of the composition described will be inherited with extensions relevant to the calculator. The basic classes are inherited to CalculatorModel, DigitView, KeypadView, ButtonView, DigitController, KeypadController and ButtonController. They perform operations specific to their responsibilities with no operations related to any object interactions. Figure 4 shows the calculator along with its basic objects.

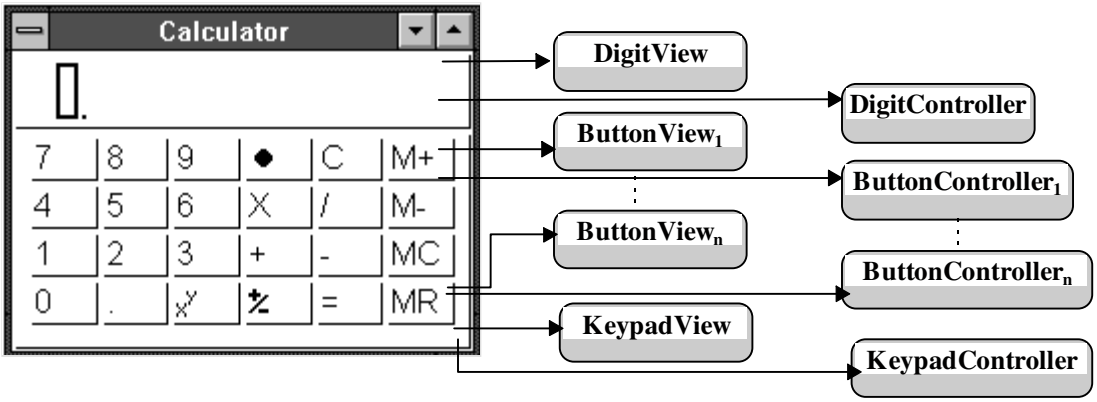


Figure 4: Calculator and its views

The interactors in the calculator application are:

- CalculatorConsistence and ViewConsistence are consistence interactors.
- DigitTranslator, KeypadTranslator and ButtonTranslator are translate interactors.
- StandardControl, DigitControl, KeypadControl and ButtonControl are control interactors.

The following figure provides an object diagram for the calculator. Note that basic MVC objects have no references to other objects, they are responsible for their own basic behavior. Only interactor objects are involved in coupling objects specific to the interaction.

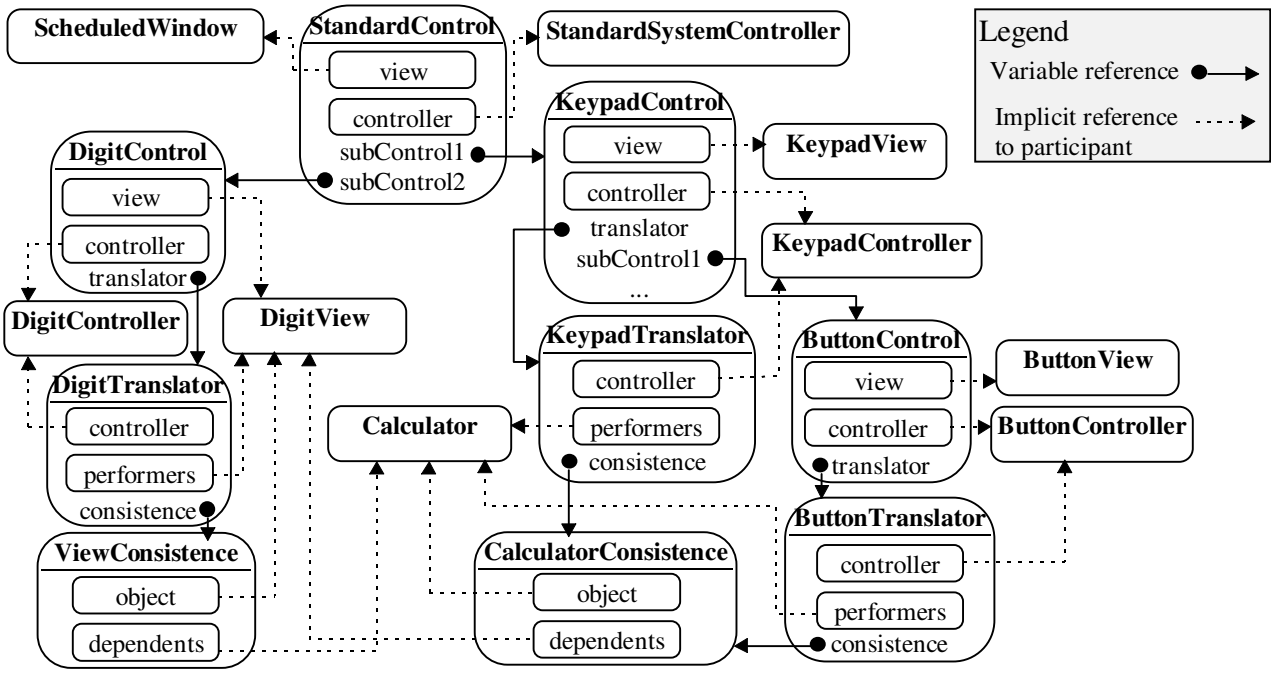


Figure 5: Calculator compositions

Note that once the implicit reference (dotted arrow lines) are linked in the interactors, the responsibilities and access to the basic objects are clearly defined. Each interactor will access only methods that they need, unlike the MVC implementation without interactors.

2.3 DynaSpecs

This research develops DynaSpecs, a specification method for describing the dynamic behavior of object interactions. The models of interactors are parameterized machine classes. An interactor defines the interaction

between two or more entities. We call these entities participants. The formal generic parameters are binded to object and/or interactor classes to create concrete interactor machine classes. The classes given as arguments constitute the participant classes.

Apart from its own attributes and methods, an interactor may require the participants to define some attributes and/or methods. These requirements are specified in the interactor's definition and conform the contractual aspects of the specification. The contractual aspects determine the minimum requirements for classes to be able to participate in an interactor. It is a contractual responsibility of the participant's classes to define their corresponding deferred attributes and methods. The methods and attributes that conform the contractual obligations are used to define the interactor's behavior.

The required attributes and methods defined by the participant classes do not need to have the names used by the interactor. In such cases a table of mapping must be provided. This table allows the mapping of names of required attributes and methods as specified by the interactor class to the corresponding attributes and methods on the participant classes, and vice versa. It is basically a translation table used by the interactors to talk to its participant entities in the language they understand.

Actual interaction behavior is defined in terms of a finite state machine. We will use StateCharts [Harel87] as the state machines. Each machine has a default transition

$$\{\text{post-condition}\} \rightarrow \textbf{State}.$$

This transition expresses the expected initial values of the attributes within in its post-condition. Other transitions have the following format

$$\textbf{State}_i \ \{\text{pre-condition}\}\ \text{message}[/\text{action}]\ \{\text{post-condition}\} \rightarrow \textbf{State}_j.$$

Pre-conditions specify the conditions that must hold for a transition to fire. Post-conditions specify the changes to attributes as a result of firing the transition. There could be transitions with no specified pre-condition. This simply indicates that the transition can be fired when the message is received. Underlined attributes refer to the value of the attributes before execution of the method on the transition. An action can be optionally specified on a transition. An action refers to a message to be sent if the transition is fired.

A transition of the form

$$\textbf{State}_i\ \text{TR}(\text{condition}) \rightarrow \textbf{State}_j$$

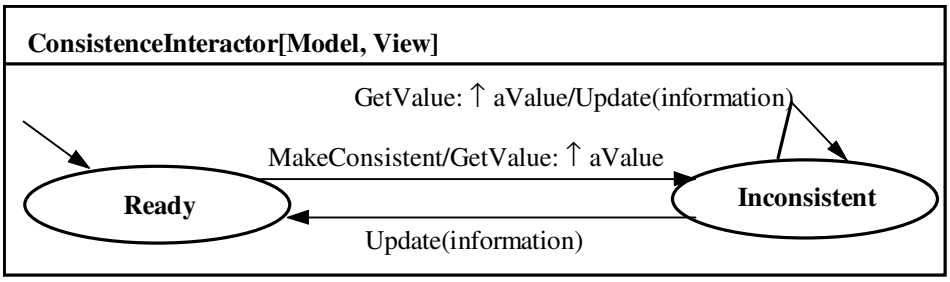
is a transition that is processed every time a transition is fired. If the condition on the TR transition is true then the corresponding TR transition is fired.

In this section we concentrate in the modeling of the consistence interactor, and the basic object classes DigitView and Calculator using DynaSpecs.

2.3.1 Modeling Consistence Interactor with DynaSpecs

DynaSpecs allows interactors to be explicitly modeled. Figure 6 shows the parameterized DynaSpecs specification for ConsistenceInteractor. It requires Model to provide the information to be displayed. It also requires the participant View to know how to update itself using the provided information to display. The consistence interactor requires View to be updated. As a result, View redisplays itself according to the Model's information provided by the interactor. This knowledge is kept and managed by the corresponding interactor, ConsistenceInteractor. Now we have an encapsulated interactor that explicitly describes the interaction between objects. All behavior and aspects of the collaboration are explicitly stated within the interactor. The effects of the collaboration upon the participating objects as well as their contributions can be found within the interactor. It is not scattered among the collaborating objects. This DynaSpec can be instantiated by providing the actual collaborating classes: Calculator and DigitView.

```
CONSISTENCEINTERACTOR[MODEL, VIEW]
                    METHODS      [MakeConsistent;]
                    ATTRIBUTES   [information;]
MODEL               METHODS      [GetValue;]
VIEW                METHODS      [Update;]
```



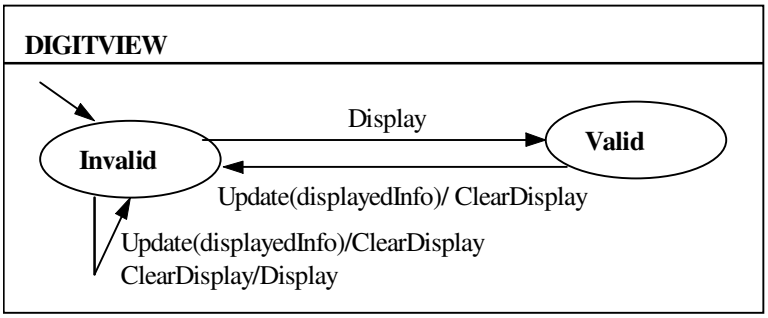
Transitions:
1. {information = NIL} → **Ready**
2. **Ready** { } MakeConsistent/GetValue: ↑ aValue { } → **Inconsistent**
3. **Inconsistent** { } GetValue: ↑aValue/Update(information) {information = aValue} → **Inconsistent**
4. **Inconsistent** {information ≠ NIL} Update(information) { } → **Ready**

Figure 6: Specification of ConsistenceInteractor

2.3.5 Modeling the Calculator Application

The specification of the class View is as shown on Figure 7.

DIGITVIEW ATTRIBUTES [displayedInfo;]
METHODS [Display; Update; ClearDisplay;]

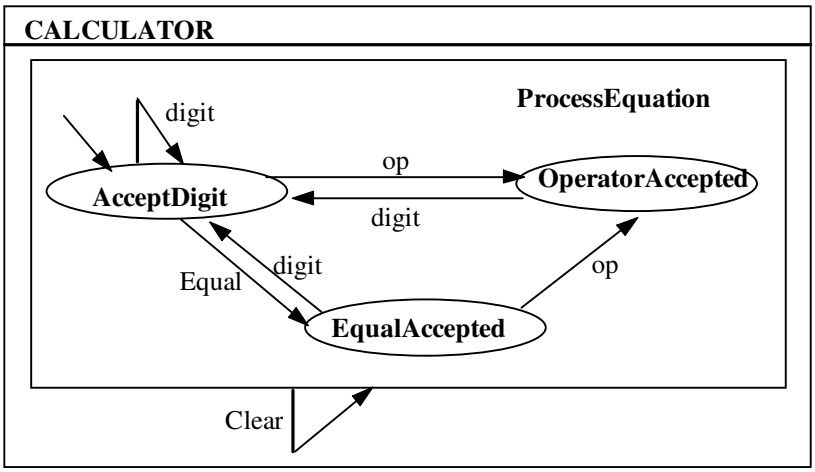


Transitions:
1. { } → **Invalid**
2. **Invalid** { } Update(displayedInfo)/ ClearDisplay { } → **Invalid**
3. **Invalid** { } ClearDisplay/Display { } → **Invalid**
4. **Invalid** { } Display { } → **Valid**
5. **Valid** { } Update(displayedInfo)/ ClearDisplay { } → **Invalid**

Figure 7: Specification of Object Class View

The object class Calculator is the actual model of the application. Figure 8 defines its behavior. Calculator accepts expressions entered in infix notation. It accepts one digit or operator at a time. Calculator provides methods that respond to each of the keys on its keypad (e.g. digits: one, two, three,... and operators: add, multiply,...). In the Calculator state machine digit ∈ {zero ... nine} and op ∈ {add, multiply, ...}.

CALCULATOR ATTRIBUTES [accumulator; operand; operator;]
METHODS [Multiply; Add; Equal; Clear; One; Two; Three; ...]



Transitions:

1. {accumulator = 0 AND operand = 0 AND operator = NIL} → **AcceptDigit**
2. **ProcessEquation** {} Clear {operator = NIL AND accumulator = 0 AND operand = 0) → **ProcessEquation**
3. **AcceptDigit** {} digit {operand = 10 * operand + digit} → **AcceptDigit**
4. **AcceptDigit** {} op {(operator ≠ NIL ⇒ operand = solve(accumulator, operator, operand)) AND operator = op} → **OperatorAccepted**
5. **OperatorAccepted** {} digit {accumulator = operand AND operand = digit} → **AcceptDigit**
6. **AcceptDigit** {operator ≠ NIL} Equal {operand = solve(accumulator, operator, operand) AND operator = NIL} → **EqualAccepted**
7. **EqualAccepted** {} digit {accumulator = operand AND operand = digit} → **AcceptDigit**
8. **EqualAccepted** {} op{operator = op} → **OperatorAccepted**

Figure 8: Specification of Object Class Calculator

Note that neither DigitView or Calculator has knowledge of their mutual existence or collaborations. They have no knowledge of the way they interact. This facilitates the reusability of these models in other applications. Similarly, the encapsulation and abstraction of interactions modeled as parameterized interactors in DynaSpecs facilitates reusability and extendibility of interactions.

A concrete interactor machine is obtained from a parameterized interactor machine by binding the parameter list with actual machine classes. A concrete interactor machine class is defined by its class name and the class names of its participant classes, the list of contractual attributes, mapping tables, and the state machines.

Let ConsistenceInteractorMachine be the parameterized interactor machine class defined previously. The object classes DigitView and Calculator are defined as follows:

DigitViewMachine = (DIGITVIEW, [displayedInfo], [Update, ClearDisplay, Display], StateMachine).
CalculatorMachine = (CALCULATOR, [operand, operator, accumulator],
[One, Two, Three, Add, Multiply, Equal, Clear, ...], StateMachine)

The binding of ConsistenceInteractor to Calculator and DigitView as actual parameters defines the concrete interactor CalculatorConsistence class

CalculatorConsistenceClass = ***BindParameterizedInteractor*** [ConsistenceInteractorMachine]
[Calculator,DigitView]
= ((ConsistenceInteractor, [Calculator, View]),
(([information]), [([makeConsistent],[GetValue], [Update]),
StateMachine)

The CalculatorConsistenceClass definition specifies that Calculator and View are actual participant classes. The semantics of DynaSpecs does not restrict the participant classes to be object classes only. They can be interactor classes as well. The only restriction is that the proposed participant classes have to conform to the contractual obligations specified by the parameterized DynaSpec.

An instance of an object class is a 4-tuple defined by a unique identifier, its class name, its memory, and its current state. A memory keeps the attributes along with their corresponding values. The state of an instance refers to its current state with respect to the state machine that defines its behavior.

A new object is created by providing the object machine class and a unique identifier. Lets create an instance of Calculator called calculator.

calculator = ***NewObject***[calculator] CalculatorMachine
= (calculator, Calculator, MEMORY, STATE)
where MEMORY = (calculator, [(operand, ⊥), (operator, ⊥), (accumulator, ⊥)] and
STATE = NIL)

The initial state of each machine component is NIL (i.e. no transition has been fired); and the initial values of the attributes in the object's memory are undefined (i.e. no values have been assigned to the attributes).

To create an interactor instance, a unique identifier for the new interactor, the interactor class that defines its behavior and the list of participants' ids must be provided. An instance interactor has a composed class name. It is formed by the interactor class name and the participants' classes. Since the Id of each participant is part of interactor instances, every interactor knows exactly who its actual participants are. An interactor cannot be created unless the actual participant entities exist. The actual consistency interactor, calculatorConsistence, is defined as follows:

consistency = ***NewInteractor***[calculatorConsistence] CalculatorConsistenceMachine
[calculator, digitView]
= ((calculatorConsistence, [calculator, digitView]), (CalculatorConsistence,
[Calculator, DigitView]), MEMORY, STATE)
where MEMORY = [(information, ⊥)], and STATE = NIL

A history is generated from a given test list [Jac92] according to the state of the entities in the environment. A test list is a list of test cases. A test case is a message sent to a particular entity within the environment. If the message has formal parameters, then actual parameters must be provided in the test case.

[(calculatorConsistence, MakeConsistent] is a test list composed of one test case. The message MakeConsistent is sent to calculatorConsistence.

Let us assume that the calculator operand value is 1, and that digitView is the Invalid state. The following is the history generated for the test list

[(calculatorConsistence, MakeConsistent, Inconsistent),
(calculatorConsistence, GetValue, Inconsistent),
(calculator, GetValue, ProcessEquation),
(calculatorConsistence, Update(1), Inconsistent),
(digitView, Update(1), Invalid),
(digitView, ClearDisplay, Invalid),
(digitView, Display, Valid)].

2.4 Compositions

This section explains the Composition language construct to implement interactors. Following the example above we will concentrate on the consistence interactor.

2.4.1 Consistence Composition

Before diving into the code, a few words about the syntax is in order. To define a composition, the programmer declares shared variables, defines the interfaces to access participating objects, and defines composition procedures and initialization statements.

```
COMPOSE Id_compose ( VariableDeclarations )
    [PARTS] CLASS Id_participant1 {…}
    …
    [PARTS] CLASS Id_participantn {…}
    Procedures
BEGIN Statements END
```

Figure 9: COMPOSE syntax

Each participant definition in the composition is a class definition. The optional word "PARTS" is used when defining a list of participants of the same participant type. For example, the consistence composition may have multiple dependents which can be defined as "PARTS" participant. The composition procedures (***Procedures***) are visible to all participants. The responsibility of each participant may be defined in the participant class or defined as abstract procedures. The abstract procedures will be replaced by procedures belonging to actual objects attached to the participants. The actual objects are bound to the participants using the RELATE clause (explain later). Therefore, only objects that define the abstract procedures can fulfill the obligation as the participant in the composition.

Now, the consistency composition code.

```
COMPOSE Consistence ()
    CLASS ModelParticipant ()
        ABSTRACT getValue ():INTEGER;
    END;

    PARTS CLASS DependentParticipants ()
        ABSTRACT setValue (newValue: ModelParticipant);
        ABSTRACT update ();
        PROCEDURE setDependentValue (newValue: ModelParticipant) =
            BEGIN SELF.setValue (newValue.getValue()) END;
    END;

    PROCEDURE makeConsistent (changeContext: INTEGER):INTEGER =
    VAR     dependentValue: ModelParticipant;
            dependents: LIST DependentParticipants;
            dependent: DependentParticipants;
    BEGIN
        dependents:= CSELF.DependentParticipants ();
        WHILE LENGTH (dependents) > 0 DO
            dependentValue:= CSELF.ModelParticipant_getValue ());
            dependent:= HEAD (dependents);
            dependent.setDependentValue (dependentValue);
            dependent.update ();
            dependents:= TAIL (dependents);
        END;
    END;

BEGIN END;
```

Figure 10: Consistence composition code

The abstract procedure "getValue" in ModelParticipant, "update" and "setValue" in DependentParticipants will be replaced when the Consistence is instantiated. Each participant may be referred to itself as "SELF" which is done in the "setDependentValue" method. The composition may be referred to itself as "CSELF" which is done in the composition procedure "makeConsistent". To refer to a specific participant procedure, it must be qualified with the participant name. For example, ModelParticipant_getValue(). The entire

list of objects for each "PARTS" participant is returned when the participant name is called. For example, calling CSELF.DependentParticipants() return the list of all dependent objects. As illustrated in figure 3, the Consistence composition object receives the "makeConsistent" message. This message will make all the calculator dependents consistent. To make them consistent, it must first get the model state by sending the "getValue" message to its model. It then send the "update" message to all its dependents.

We instantiate and relate objects using the RELATE clause. Objects are related by passing actual object as arguments. The syntax for the RELATE clause is

$$\textit{Id}_{\textit{compose object}} = \text{RELATE } \textit{Typename}_{\textit{compose}} (\textbf{object}_1, \ldots, \textbf{object}_n)$$

Figure 11: RELATE syntax.

The object arguments may be a list of objects if they are bind to "PARTS" participants. The following code instantiate a calculator's Consistence composition and relate its model and dependents to specific participant objects.

```
digitView:= NEW (DigitView);
calculator:= NEW (Calculator);
calculatorConsistence:= RELATE CalculatorConsistence (calculator, [digitView]);
```

Figure 12: Instantiating calculator consistence composition

CalculatorConsistence is a subCompose of Consistence composition. It includes code specific to maintaining the consistency between a display view of the calculator call digitView and its model, the calculator. Calculator is the model and digitView is the dependent of calculatorConsistence. The model, calculator implements and replaces the abstract procedure "getValue". The dependent, digitView implements and replaces the abstract procedures "update" and "setValue". Since the DependentParticipants is a "PARTS" declaration, there may be more than one dependents. The binding of dependents with one element in a list, the digitView, instantiate only one "PARTS" participant.

2.4.2 Calculator Compositions

The translate and control compositions are implemented similar to the consistence compositions. They define the abstract design pattern for the interaction. Each composition is then subComposed to include application specific code to implement the calculator. They are the CalculatorConsistence and ViewConsistence as subCompose of the consistence composition, DigitTranslator, KeypadTranslator and ButtonTranslator as subCompose of the translate compositions, and StandardControl, DigitControl, KeypadControl and ButtonControl as subCompose of the control composition.

The following code instantiates and relates all objects in a calculator.

```
keypadView:= NEW (View);
keypadController:= NEW (KeypadController);
digitView:= NEW (DigitView);
digitController:= NEW (Controller);

calculator:= NEW (Calculator);
calculatorConsistence:= RELATE CalculatorConsistence (calculator, [digitView]);
viewConsistence:= RELATE ViewConsistence (digitView, [calculator]);

digitTranslator:= RELATE DigitTranslator ([digitView], digitController);
digitTranslator.PerformerParticipants ().initializeConsistence (viewConsistence);
digitControl:= RELATE ControlInteractor (digitController, digitView);
digitControl.ControllerParticipant_initializeTranslators ([digitTranslator]);

keypadTranslator:= RELATE KeypadTranslator ([calculator], keypadController);
```

```
keypadTranslator.PerformerParticipants ().initializeConsistence (calculatorConsistence);
keypadControl:= RELATE KeypadControl (keypadController, keypadView);
keypadControl.ControllerParticipant_initializeTranslators ([keypadTranslator]);
keypadControl.buildButtons ([calculator], [calculatorConsistence]);

calculatorView:= NEW (View);
calculatorController:= NEW (Controller);
calculatorControl:= RELATE ControlInteractor (calculatorController, calculatorView);
calculatorControl.appendSubCInteractors (digitControl);
calculatorControl.appendSubCInteractors (keypadControl);
calculatorControl.startUp ();
```

Figure 13: Instantiating calculator compositions

To reduce the code, we omitted the instantiation and relation of each Button. They are done in the procedure buildButtons ([calculator], [calculatorConsistence]). Note the similarity of instantiating and relating digit and keypad. One difference is the performer for their translator. The performer for digitTranslator is a digitView while the performer for keypadTranslator is a calculator. Another difference is the consistence. The consistence for digitTranslator is viewConsistence while the consistence for keypadTranslator is calculatorConsistence. Note that control passes its translator to its controller and translate passes its consistence to its performer. The calculatorControl sets up the view hierarchy using "appendSubCInteractors" and start up using "startUp".

3 Conclusion

This paper presents a unified approach to model and implement interactions as first class objects. During analysis and design, they are modeled using DynaSpecs. During implementation, they are coded with a new language construct called Compositions. Both DynaSpecs and Compositions provide a consistent support for object interactions within each phase of the OO lifecycle.

Making interactions first class citizens within OO systems requires support during all phases of software development. Our research has already developed a rich support environment for such interactions from analysis to implementation. Interactions are specified explicitly using DynaSpecs, a formal method for describing the dynamic model of object interactions. DynaSpecs model the object interactions using state transitions. The state transitions represent events of message calls that mirror the dynamic behavior of object interactions. The formal semantics of the model is based on the idea of a history which captures the sequence of operations within interactions along with state changes undergone by the participating objects. A major attraction of this model is that it is effective not only in the variable degree of abstraction of interactions among entities but also in the accurate representation of the application environments. This research defines the formal semantics for one and only one interaction environment. Allowing an application to have more than one interaction environments is appealing. Objects could move from one interaction environment to another within the same application. Each environment defining a context of interaction allows objects to have completely different sets of interactions in different environments within the same application.

The Composition construct clearly defines object interactions as first class values. Being first class values, interactors, through the composition construct and DynaSpecs, reduces complexity, loosens coupling, improves reusability of object interactions, and allows better abstraction of interactions. The main properties by interactors are

- definition of participants and their responsibilities,
- defining proper objects suitable to be participants through abstract procedures,
- restricted visibility to participant objects to only methods relevant to the interaction,
- reuse of interactions by plugging participants or inheriting compositions,
- providing a dynamic lifetime of an object interaction, and
- providing the necessary context to define the interactions.

The case study demonstrate how to capture object interactions using DynaSpecs and Compositions. The calculator provides an example of reusing the compositions. Prototype executable of both construct have been

implemented. The semantics of the prototypes are defined in denotational semantic. They depict the precise behavior of the construct and all its relevant context and scope. The one-to-one relationship between DynaSpecs developed during analysis/design and Compositions in the implementation provide a consistent approach. Thus, changes to the system can be made to DynaSpecs and translated to Compositions. This approach makes it easier to establish consistency between the analysis/design phase and the corresponding implementation and simplify reasoning and validating the modeled Object-Oriented software.

References

[Boo94] Booch Grady. Object-Oriented Analysis and Design with Applications, Second Edition. The Benjamin/Cummings Publishing Company, Inc. 1994.

[Cole94] D. Coleman et al. Object-Oriented Development: The Fusion Method. Prentice Hall, 1994.

[Doda94] Dodani Mahesh and Gan Kok Siew. A Semantic Framework for Understanding the Behavior of Modules and Classes in Programming Languages. Proceedings of the Joint Modular Language Conference, University of Ulm, Germany, ISBN 3-89559-220-X, pages 79-92, 28-30 September, 1994.

[Gam95] Gamma Erich, Helm Richard, Johnson Ralph and Vlissides John. Design Patterns: Elements of Object-Oriented Software Architecture. Addison-Wesley Publishing Company, 1995.

[Jac92] Jacobson Ivar, Christerson Magnus, Jonsson Patrik and Overgaard Gunnar. Object-Orietnted Software Engineering: A Use Case Driven approach. ACM press, Addison-Wesley Publishing Company, 1992.

[Harel87] D. Harel et at. On the Formal Semantics of StateCharts. Symposium on Logic in Computer Science. Ithaca, N.Y. June 22-25, 1985, pp. 54-64.

[Kam88] Kamin Samuel. Inheritance in Smalltalk-80: A Denotational Definition. Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages, pages 80-87, Jan 13-15, 1988.

[Kras88] Krasner Glenn E and Pope Stephen T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, Volume 1, Number 3, pages 26-49, 1988.

[Mod88] Model Mitchell. The Model-View-Controller (MVC) paradigm for user interfaces. Tutorial 6A of OOPSLA, September 25-30, 1988.

[Parc92] ParcPlace Systems. VisualWorks - ObjectWorks Smalltalk User's Guide, Release 4.1. ParcPlace Systems, Inc. 1992

[Red88] Reddy Uday S. Object as closures: Abstract Semantics of Object Oriented Languages. Proceedings of ACM Conference on Lisp and Functional Programming, pages 1-19, July 1988.

[Rum91] J, Rumbaugh et al. Object-Oriented Modeling and Design. Prentice Hall, 1991

[Tan95] Tanzer Christian. Remarks on Object-Oriented modeling of associations. Journal of Object-Oriented Programming, pages 43-46, February, 1995

[Tsai92] Tsai Chung-Shin. ACTS: A formal model for reliable object-oriented programming based on abstract and concrete classes. Ph.D. thesis, University of Iowa, May 1992.

[Whi94] White Iseult. Rational Rose Essentials, using the Booch Method. Benjamin/Cummings Publishing Company, Inc, 1994.

[Xav93] Xaviar Leroy. The Caml Light system, release 0.6. Documentation and user's manual. ftp.inria.fr (128.93.1.26), September, 1993.