Event-based Implicit Invocation Decentralized in Ada*

LIANG Xianzhong, WANG Zhenyu
E-mail: {liangxz, zywang}@public.wh.hb.cn
Wuhan Digital Engineering Institute
P. O. Box 74223, Wuhan, P. R. CHINA

[Abstract] *Nowadays more and more attraction is drawn by the event-based implicit invocation – one of useful architectural patterns, because of its loose coupling between components in the architecture and reactive integration in software systems. Analyzing object-oriented interaction with objects, this paper, based upon the principle of software architecture, presents an approach on event-based object model with Ada exception handler. Consequently it is possible for us to improve, with adding specific architectural patterns, traditional programming languages into architectural description languages.*
Key words: Implicit invocation, Architectural pattern, OO (object-oriented), EBO (event-based object), Exception handler

A. Introduction

Software development is shifting its focus from lines-of-code to coarser-grained components. Software architecture has been proposed to respond such a high level design that invokes the *elements* composing systems, *interactions* among those elements, *patterns* guiding their composition, and *constraints* on these patterns [1, 2].

The most common interactions among components are *procedure call* and *shared data* that are directly supported almost by all programming languages. However, interactions become more and more complicated, such as the *access to database*, *internetworking*, *concurrent & distributed communication* and event-based *implicit invocation* and so forth. Unfortunately most programming languages cannot afford these kinds of interactions unless applying some specific facilities supported by special software products or operating systems. For instance, ODBC is designed for database connectivity.

From OO to EBO

Some specific programming methodologies or languages have limited capabilities in some extent to which the interactions in systems are determined. E.g., The components developed by functional decomposition can only interact through procedure call or shared data; and the objects in OO systems can interact through message-sending -- *explicit invocation*. This paper aims at introducing event-based interaction in OO methodology, so that an OO system can be evolved to a new-patterned system. The new-patterned system is known as an EBO system, which mainly concerns event-based interaction between objects. And then, the objects in EBO systems can interact through event-announcing to triggers *implicit invocation* to the other objects' routines.

Why EBO systems are needed?

As well known, there are many advantages in OO systems, such as modularity, data encapsulation and intuitive representation of domain knowledge to solve software problems, but explicit invocation has some disadvantages. The most significant is that an object interacts with another (via message-sending) must know the identity of that other object[1]. So changing the identity of objects would result in whole system's "quake" (re-compilation).

Implicit invocation in EBO systems provides an excellent interactive way, which de-couples interconnection between objects. In principle, an object announces (or broadcast) one or more events. Other objects in the system can register an interest in an event by associating one of routines of these objects with it. When announcing an event, the EBO system itself "implicitly" causes the invocation to the associated routines. Consequently, early-developed objects can make an active interaction with later-developed objects without re-compiling the whole system, which means reactive integration of EBO systems.

As one of valuable architectural patterns, event-based implicit invocation not only supports loose coupling between interactive objects, but also complements the fault of high coupling in OO systems with reactive integration. So EBO systems with coarser-grained components are expected to have following attractive properties:

- **Replaceable component.** An object hides the complexity into a black-box and implicit invocation makes the object replaceable without producing any blight or "quake" on the systems.
- **Reusable architecture.** The generalized system of DSSA (domain-specific software architecture) has highly abstracted structure and reusability, so specialized systems can be derived by means of replacement of some specific components.
- **Piecemeal growth.** Through announcing events, early-developed objects trigger invocation to reactive routines of later–developed objects, which makes the compositional system evolved repeatedly and incrementally.

Current state

Many new methodologies and operating platform came into the world in spite of different identification. The broad-accepted concepts are *event-driven programming* in Windows and *component-based development* in JavaBeans [3]. Because of

* Supported by National Natural Science Foundation of China

well-defined event mechanism, Windows has become the famous framework on which many useful products are developed, such as, Visual C++ and COM and so on. However, the *centralized* event mechanism in Windows is very complex and lacks for flexibility -- no one wants to tie his products with the specific platform without portability. In contrast, JavaBeans derived from Java is platform-independent, which attractively permits to build *decentralized* event source and triggering mechanism by designer, so that dependency of event handling is dramatically decreased. But "write once and run anywhere " that is promoted by Java would not satisfy the needs in developing embedded systems.

Ada has been broad used for large-scale, embedded real time, mission-critical and high reliable systems. Several researches have been made for event-based handling object. ObjectAda for Windows is a successful product that maps object-orientation in Ada95 into Microsoft Foundation Classes (MFC). Unfortunately the successful ObjectAda sacrifices itself to portability. The other research is from ADL domain that adapts traditional programming languages into ADLs by adding implicit invocation to Ada [1]. In this research, a package is treated as an interactive element and implicit invocation is implemented through hiding the identity of the package from client packages. However a package is only a static design structure, which is neither an object that can be polymorphically reattached nor a functional module that can be dynamically bound with an access interface, so the composed system does not permit to replace one package with another *at run time*.

Our effort

Since ADLs need developing in their applicability, to adapt traditional programming languages to ADLs may be a significant approach. Obviously full adaptation from a programming language to an ADL is very difficult because the big gap between them needs bridging, but partial adaptation is possible, for instance, let Ada -- the most excellent programming language to be enhanced with specific architectural patterns. Formerly we have done the enhancing work on Ada through CASE Tool [4], which is considered valuable.

In supplement section, the object-oriented concepts in Ada are outlined, which is essential for further discussion. In section B, an EBO model in Ada is originally presented, which is used to construct event-based implicit invocation decentralized in Ada. In section C, the dynamic behavior of EBO model is discussed, which fully exploits Ada exception handler. At last a compositional EBO application in Ada is also given.

Supplement: Object-orientation in Ada

Through the historical development of Ada83 and Ada95, Ada has enhanced capabilities of supporting full object-orientation [5]. The package in Ada is the most excellent programming unit that fulfils ADT design. Especially in a package, the type of ADT defined with tagged and controlled properties can be extended. However, a package is only a static design structure instead of pristine class construct. We need adapt a record type into a class that fully encloses features, including attributes and routines, which well reflects important object-oriented concepts, and is known as ***A-ObjAda*** [6]. A-ObjAda, an Ada-based class description language, is characterized as following:

- Class and encapsulation

A class encloses its attributes and routines (mainly concerning the interface). The former represents fields of the objects of the class and the later represents computations on these objects (see Example A-1: *Adapted class*).

- Inheritance hierarchy

Through extension of the adapted class we can build inheritance hierarchy. With a more generalized class *Person*, a more specialized class *Teacher* is derived with extension (see Example A-1: *Inheritance hierarchy*)

- Dynamic binding & Polymorphism

Being derived from *Person*, *Teacher* not only inherits Person's features, but also adds new features for itself, and redefines routine *Display* through overriding. The overridden routine means different versions will share the same interface. Upon that the interface is bound with the related version at run time by object constructor, which reflects polymorphism (see Example A-1: *Polymorphism*).

-- Example A-1: Object-orientation in A-ObjAda

Adapted class	Inheritance hierarchy	Polymorphism

class Person **is**
  procedure Display **is interface**;
  procedure Set-Name(n: **in** tName) **is interface**;
  function Get-Name **return** tName **is interface**;
  procedure Set-Sex(s: **in** tSex) **is interface**;
  function Get-Sex **return** tSex **is interface**;
  procedure Set-Age(a: **in** tAge) **is interface**;
  function How-Old **return** tAge **is interface**;
private
  Name **:** tName;
  Sex **:** tSex;
  Age **:** tAge;
end Person;
	class Person **is**
	…
	end Person;
	class Teacher **is new** Person
	  procedure Display **is overridden**;
	  procedure Set-Course(C: **in** tCourse) **is interface**;
	  function Get-Course **return** tCourse **is interface**;
	  procedure Set-Unit(C: **in** tSchool) **is interface**;
	  function Get-Unit **return** tSchool **is interface**;
	private
	  Speciality **:** tCourse;
	  Unit **:** tSchool;
	end Teacher;
		procedure Main **is**
		  pObj : aPerson := **new** Person;
		  tObj : aTeacher := **new** Teacher;
		begin
		  …
		  pObj<- Display;
		  --> Person's version applied
		  pObj := aPerson(tObj);
		  -->polymorphic reattachment
		  pObj<- Display;
		  --> Teacher's version applied
		  …
		end Main;

Where *aPerson* and a*Teacher* is *general access* to class Person and Teacher, respectively. In the inheritance hierarchy, a top general access can legally refer to any object whose class is descendant from the top class, so polymorphism is implemented by type casting reattachment. Thus, we have discussed, in straight and concise form, the characteristics of object-orientation in A-ObjAda, which is useful to describe and implement EBO model.

B. Event-Based Object Model

In order to support reactive integration of EBO systems, an object should have triple capabilities: *event-announcing*, *event-exchanging* and *event-listening*. The first aspect denotes that an active object announces an event so as to trigger an implicit invocation of the other object's routines. The second aspect administers the communication between interactive objects during invocation, and the last aspect denotes that the passive object as listener registers an interest by associating one of its routines so

as to accept the invocation.

Sketch of EBO model

EBO model is composed of tree elements: An active class whose objects announce events and then triggers implicit invocation to other object's routines, a passive framework treated as an abstract class, from which listening classes are derived so as to listen to the announced events, and an event construct that provides a standard format for the communication between interactive objects. **Fig. 1** outlines the sketch of EBO model.

The object of active class (*announcer* for short) announces an event only against the passive framework and the object of the derived class from the framework (*listener* for short) listens to the event. Once an event announces, the listener's routine will be immediately invoked. Because the passive framework separate the announcer from the listener, the invoked routine does not need an explicit identity of that listener (object), which effectively de-couples interconnection between announcer and listener.

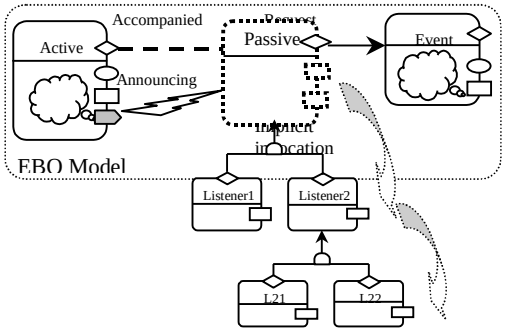

Fig.1 Essential elements of EBO model and their relationship

Event construct

The event construct provides the standard format for the communication between interactive objects, while a variant record in Ada can be designed for multiplex events. For the identification of multiplex events, a discriminant treated as the key code is used to construct a parameterized class. E.g., the class dealt with two formats of information (see Example B-1).

-- Example B-1: *Event structure*

Event structure

```
package Events_P is
  type Event_Key is(co_Event1, co_Event2);
  ex_Event1 : exception; -- > co_Event1
  ex_Event2 : exception; -- > co_Event2
  class Events(Code : Event_Key) is
    Item1 : Type1;
    Item2 : Type2;
    case Code is
          when co_Event1 => Item11 : Type11;
          when co_Event2 => Item21 : Type21;
      end case;
    end Events;
end Events_P;
```

Three important aspects are involved in the construct:

- **Event codes.** They are used to identify different events, for an announcer can have multiple events. E.g., we have defined two events, which are denoted by co_Event1 and co_Event2, respectively.
- **Event exceptions.** Ada exception is used to immediately raise some actions, which is exploited here as the trigger of implicit invocation. Two event exceptions are defined as ex_Event1 and ex_Event2 corresponding to the Event_Key.
- **Variant structure.** It is used for representation of multiple events. A discriminant allows variant data items enclosed in the class, e.g., *Events*(co_Event1) represented by such items as (Code, Item1, Item2 and Item11), while *Events*(co_Event2) by (Code, Item1, Item2 and Item21), which is essential for information-passing duration event-based interaction.

Passive framework

Passive framework is an abstract class with a set of abstract reactive routines, which provides interfaces for reactive routines. Announcing an event will trigger invocation to these interfaces, and the listening class derived from the passive framework overrides interfaces by giving substantial implementations. The passive framework needs to provide an abstract routine for each of events, which is described in Example B-2.

-- Example B-2: *Passive framework*.

Passive Framework

```
with Events_P; use Events_P;
package PassFrame_P is
  abstract class PassFrame is
      procedure On_Event1(E: in out Events) is abstract;
      procedure On_Event2(E: in out Events) is abstract;
  end PassFrame;
  type aPassFrame is access all PassFrame;
```

```
end PassFrame_P;
```

Several important concepts need discussing in the passive framework:

- **Abstract framework.** The framework only provides listening interfaces without any substantial implementation, so the derived classes from the framework are expected to override the related routines. The objects of the derived classes treated as listeners of events will give immediate and substantial reaction when events announced.
- **Parameterized event.** All of abstract reactive routines in the framework need an explicit argument, which is used for communication between interactive objects. As stated before, the argument containing discriminant item can fulfil information–exchanging of multiple events.
- **General access.** General access provides a top access interface used for polymorphic reattachment of future listeners, i.e., being, through type casting, reattached with a listener, this top access can select the overridden routines to be invoked.

Active class

An active class is the motive force that raises the invocation, whose objects (announcers) will forwardly interact with other objects by means of announcing events. a given announcer needs to provide a set of essential facilities for both event-based interaction and duty functionality (see Example B-3).

-- Example B-3: *Active class*

Active class

```
package ActiveEvent_P is
  class ActiveEvent is
    procedure Reg_Event1 (Listener : aPassFrame) is interface;
    procedure Reg_Event2 (Listener : aPassFrame) is interface;
    procedure Announce_Events(E: Events) is interface;
    procedure Trigger_Invocation(E: Events) is interface;
    procedure Action is interface;
  private
    Event1_Listener : aPassFrame;
    Event2_Listener : aPassFrame;
  end ActiveEvent;
end ActiveEvent_P;
```

- **Registering services.** Listeners apply them to register their interest in events, through which the announcer will record who provides substantial reaction for the given event. Obviously listeners are held by attributes *Event1_Listener* and *Event2_ Listener* in class ActiveEvents.
- **Event-handling services.** They are responsible for the communication, interpretation of events, and trigger of implicit invocation. When announcing an event, the announcer forwardly raises, with the registered listener, invocation to the concrete routine depending on the event code
- **Specific functionality.** Besides providing essential facilities for motivation of interaction with listeners, an announcer should exercise its normal duty, say specific functionality the component is designed for, which is implemented as *Action* procedure.

Listening hierarchy

As discussed before, the passive framework provides the abstract listening framework, from which multiple listeing classes can be derived so as to build a (listening) inheritance hierarchy "tree". Besides normal duty of its own, every listening class is free to override specific reactive routines, so that the class's objects give polymorphic reaction for some events. Such a special listening class hierarchy can be built in a way. E.g., *Listener2* is *PassFrame*' s immediate subclass and overrides routine *On-Event1*, whose objects are only interested in the Event1, while *L21* is *Listener2*' s subclass and overrides routine *On-Event2*, whose objects are interested in both Event1 (inherited from *Lister1*) and Event2, described as Example B-4.

-- Example B-4: Listening hierarchy

Passive class

```
class Listener1 is new PassiveFrame
   … …
end Listener1;
----------------------
class Listener2 is new PassiveFrame
   procedure On_Event1(E: in out Events) is overridden;
   …
end Listener2;
----------------------
class L21 is new Listener2
   procedure On_Event2(E: in out Events) is overridden;
   …
end L21;
```

In principle, the passive framework separates the announcer from the listener, so the invocation to reactive routines does not need direct identity of the concrete listener, which is known as implicit invocation. In this way, EBO model effectively de-couples interconnection between announcers and listeners. When a listener is replaced with another one, the announcer does not care -- without needing re-compilation.

C. Event-Based Exception Handling

Ada exception handler perfectly separates exception detection from exception handling. During program execution, with detecting a special case, say a mistake, an exception is immediately raised, which transfers the control of program to predefined exception handling section. And after handling the exception, the control of program resumes its execution.

Event-based object model and the relatively-defined facilities are only described by static design construct, which does not seem to reflect real interaction, for the behaviors of event-announcing and immediate reaction dynamically occurs at run time. Here we will exploit the excellent Ada exception handler to implement.

Trilogy of implicit invocation

In some significance, that raising an exception results in control transferring well fits in with event-based implicit invocation, which is discussed below. Example C-1 gives detailed implementation about the routines enclosed in class *ActiveEvent*.

1) Event-listening

In section B, class *ActiveEvent* has defined a set of registering services, which are used for Listener's interest in an event. This kind of routines takes general access to passive framework as an argument, which can pass the access of any listeners in the listening hierarchy to the announcer by type casting reattachment. Simultaneously class *ActiveEvent* has also provided two attributes: *Event1_Listener* and *Event2_Lisnter*, which are used to hold current listeners. The held attributes denote which listener is interested in the related event (see Example C-1: *Event-listening*).

2) Event-announcing

Ada provides many useful pre-defined exceptions. For instance, when a number is divided by zero, Ada exception handler will automatically raise the exception NUMERIC_ERROR. Also Ada permits a user to define special exceptions for his own, which may not always concern an error. Once the exceptional case occurs, he can control his program to raise the self-defined exception.

As discussed before, an event exception in EBO model is associated with an event code. In general, event-announcing needs to do two things: interpretation and announcement. The former may do some standardizing or processing work on the information of events (omitted in this paper), and the later raises the event exception, which denotes a substantial announcement of an event (see Example C-1: *Event-announcing*).

3) Invocation-triggering

As well known in object-oriented philosophy, the top general access can reference any objects, if only whose class is descendant from the top class. *Event1_Listener* or *Event2_Listener* is such a general access that is used to hold any listeners. The reactive routine overridden by any listeners, say *On_Event1*, can be invoked through this general access (see Example C-1: *Invocation-triggering*).

Once an event exception is raised, the control of program will terminate the related routine and is transferred to pre-defined exception handling section, where invocation is triggered according to the identity of event exceptions. Obviously, reactive routines are invoked in the implicit way, for the announcer does not concern which listener is referenced with the top general access.

Composition of EBO

In application, the announcer as the source of events concurrently performs its specific functionality, where events are continually announced in some regulated strategies. The routine *Action* defined in class *ActiveEvents* well embodies this idea by means of a task. In order to apply EBO model, besides building a listening hierarchy that reflects different reaction against events, a compositional unit is needed to compose an application of EBO system. Example C-2 describes this idea.

-- Example C-1: Trilogy of implicit invocation

Event-listening	Event-announcing	Invocation-triggering
procedure Reg_Event1(Listener : aPassFrame) **is** **begin** **if** Event1_Listener /= Listener **then** Event1_Listener := Listener; **end if ;** **end** Reg_Event1; **procedure** Reg_Event2(Listener : aPassFrame) **is** **begin** **…** **end** Reg_Event1;	**procedure** Announce_Events(E: Events) **is** **begin** **…** -- > interpret events **case** E.Code **is** **when** co_Event1 => **raise** ex_Event1; **when** co_Event2 => **raise** ex_Event2; **end case**; **end** Announce_Events;	**procedure** Trigger_Invocation(E: Events) **is** **begin** **… …** *Announce_Events(E);* **exception** **when** ex_Event1 => Event1_Listener<-On_Event1(E); **when** ex_Event2 => Event2_Listener<-On_Event2(E); **end** Trigger_Invocation;

There are some comments about *Action* and *Configuration* procedures:

Firstly, when triggering an invocation, the announcer (*Actor*) only concerns the current event (*E1*) without mentioning the identity of any listeners and even of the reactive routine, which embodies the full implicity of invocation, e.g.:

Trigger_Invocation(E1);

Secondly, which reactive routine is invoked only depends on dynamic reattachment of different listeners, which reflects important concepts: *Polymorphism* and *dynamic binding*, e.g.:

aReg := aPassFrame(L1'**access**);

Thirdly, in order to build an event-based interaction between an announcer and a listener, the compositional program only concerns announcer, listener and listener's interest in the event, which is so simple, e.g.:

Actor<- *Reg_Event1* (aReg); -- aReg:= aPassFrame(L1'**access**);

-- Example C-2: Active duty and compositional application

Active duty	Compositional application

package body ActiveEvents_P **is**
  task type tRun;
  procedure Action **is**
  Running : tRun;
  begin -- > here start the task *Running*
  null;
  end Action;
  task body tRun **is**
  E1 : Events(co_Event1);
  E2 : Events(co_Event2);
  begin
  loop
  … --> do specific functionality
  E1 := … --> Event information prepared
  Trigger_Invocation (E1);
  …
  E2 := … --> Event information prepared
  Trigger_Invocation (E2);
  end loop;
  end tRun;
end LactiveEvents_P;
	with ActiveEvents_P, PassFram_P, Listerner1_P, Listener2_P;
	use ActiveEvents_P, PassFram_P, Listerner1_P, Listener2_P;
	procedure Configuration **is**
	  Actor : ActiveEvents;
	  L1 : Listener1;
	  L2 : Listener2;
	  aReg : aPassFrame; -- top general access
	begin
	  -- L1 is registered for an interest in Event1 ---
	  aReg := aPassFrame(L1'**access**);
	  Actor<- Reg_Event1(aReg);
	  …
	  -- L2 is registered for an interest in Event2 ---
	  aReg := aPassFrame(L2'**access**);
	  Actor<- Reg_Event2(aReg);
	  …
	  -- Concurrent interaction to be reiterated ---
	  -- •. Do specific functionality
	  -- • Interaction between Actor and L1 through Event1
	  -- •. Interaction between Actor and L2 through Event2
	  Actor<- Action;
	end Configuration;

D. Conclusion

From the points of view of architectural pattern, a specific EBO system is collection of announcers and listeners. Passive framework effectively separates announcers from listeners which largely de-couple interconnection between objects, so listeners can be freely replaced by other objects without affecting the announcer.

EBO model rationally distributes such complexity as variant structure for events, abstract reactive routines and event exception detection in the related packages (Events_P, PassFrame_P and ActiveEvents_P), which perfectly embodies the principle of information-hiding.

Event-based implicit invocation in EBO systems is considered as an excellent architectural pattern and enhanced in Ada by means of the capacities in Ada95. On the basis of that, this paper originally presents a good solution to following problems:

- build a conceptual framework for event handling with Ada;
- present an decentralized EBO model and design methodology;
- fulfill implicit invocation by exploiting Ada exception handler .

As accompanied situation, this approach is confronted with some problems that needs to be solved in the future:

- **event broadcasting**. an event associated with a register service only permits one listener to register its interest, which means that an event can only be announced to a specific listener without broadcasting to several listeners;
- **multiple event sources**. an event source (ActiveEvents) is accompanied with the passive framework that derives a listening hierarchy (including many listeners), but for more than one event source, a listening class would be derived from multiple passive frameworks, which needs multiple inheritance to support.

For above-mentioned problems, a future solution is expected as following:

Class *ActiveEvents* should provide a listening list for every event to register several listeners. When broadcasting an event, the announcer can traverse the list so as to trigger multiple invocations.

Based on building the listening hierarchy from the passive framework, EBO model needs multiple inheritance to support that a listener listens to multiple event sources, which is limited by single inheritance in Ada95. Luckily, the passive framework only concerns abstract routines without involving any details, if necessary, multiple passive frameworks can be united as a more generalized framework, say *GPassFrame*. And then, any listeners derived form *GPassFrame* can listen to multiple event sources.

E. References

[1] Mary Shaw and David Garlan, Software Architecture: Perspectives on an emerging discipline, Prentice-Hall International, Inc1996.

[2] Nenad Medvidovic and Richard N. Taylor, A Classification and comparison Framework for Software Architecture Description Languages, IEEE Transactions on SE, VOL.26, NO. 1,Jan. 2000.

[3] Laurence Vanhelsuwe, Mastering JavaBeans, SYBEX Inc., 1997.

[4] Liang Xianzhong and Wang Zhenyu, Ada-based Support for Abstraction, Encapsulation and Unit Hierarchy, Proceedings of Tri-Ada'91 International Conference, San Jose, USA, 1991.10. P116-125

[5] Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall International, Inc1997.

[6] Liang Xianzhong, Wang Zhenyu, Remolding Diversified Objects in Ada95: Toward A-Object Pattern, Proc. ISES-01, Wuhan, China, Mar. 23-28, 2001