

Active Objects Provide Robust Event-driven Applications

Claude Petitpierre
Teleinformatics Laboratory, EPFL
Lausanne, Switzerland

Anton Eliëns
Faculty of Sciences, Vrije Universiteit
Amsterdam, The Netherlands

Abstract *The non-determinism inherent in event-driven systems encompassing both networked applications and interactive applications, makes these applications difficult to develop and maintain, despite the availability of powerful libraries. One reason for this is the so-called inversion of control needed to dispatch events to listener objects provided by the application. In this paper we will argue that event-handling with listener objects leads to problems in the applications that may be avoided by resorting to blocking calls and to active objects. The latter concepts provide a uniform approach to robust event-handling, both with respect to the registration and (de)activation of listener objects, as well as the dispatching of events to blocking operations. We will illustrate our approach by discussing how to develop a graphical user interface for an application that must access a remote database. We will also indicate that as a benefit, from a software engineering perspective, our approach leads to a more modular approach to event-based systems and a better insight in the flow of control between listeners, network accesses and the application.*

Keywords: Concurrency, multi-threading, event, callback, rendezvous, active object.

I. INTRODUCTION

THE majority of applications fall within the class of event-driven systems, encompassing both networked applications and interactive applications with a graphical user interface. The non-determinism inherent in such systems makes these applications difficult to develop and maintain, despite the availability of powerful libraries. The problem is aggravated when multi-threading is needed, for example to service multiple clients or preventing the application to block when accessing a remote resource.

This work has been supported by the EU, within project C-Arctic.

In this paper we propose an approach to event-handling based on active objects and rendezvous. The structure of the paper is as follows. In section II, we will reconsider the various way event-handling is approached, and we will indicate the problems that may occur with respect to modular code development and multi-threading. In section III, we will provide a brief description of an extension of Java that supports active objects and synchronous communication by rendezvous. In section IV, we will provide an example that illustrates how synchronous communication allows a more direct approach to event-handling. In section VI, we will present some related work. In section V, we will discuss the benefits of our approach from a software engineering perspective, and in section VII, we will draw some conclusions.

II. EVENT HANDLING RECONSIDERED

Event-driven computation underlies many applications, ranging from graphical user interfaces to systems for discrete event simulation and business process modeling. An important characteristic of event-driven computation is that control is relinquished to a library that waits for events to occur. Each event is then dispatched to the application by invoking a handler function or a handler object for appropriate action. When we reconsider how events may be handled, we can distinguish between:

- event loops – explicit dispatching on type of event
- callback functions – implicit (table-based) dispatching
- listener objects – callback on objects with hook methods

Event loops are typically the most primitive way to deal with events. Within an event loop

an explicit dispatch on the type of the event is needed to invoke the appropriate application code.

Callback functions are a significant improvement over plain event loops. The dispatching is implicit, based on an association between a callback function and the type of event. Note that this approach is more modular and new callback functions may easily be associated with the events.

Listener objects, as for example used in the Java AWT and Swing GUI libraries, may be regarded as an extension of callback functions, providing an object and a hook method that is invoked on the occurrence of an event. Since listener objects can maintain state in-between (hook) method invocations, listener objects are strictly more powerful than callback functions that must rely on ad hoc mechanisms to take the history of event occurrences into account.

The *Reactor* pattern, described in [1], provides a good example of an approach to handling a multitude of events. Operationally, the *Reactor* approach amounts to the following steps:

1. register handlers for particular event types
2. start event loop and dispatch incoming events
3. for each event, locate handler and invoke hook method
4. within handler, take action based on state and event information

Characteristic for both callback functions and listener objects is an *inversion of control*, namely the application code must wait for invocations by the (event) dispatching mechanism. Although this is definitely an improvement over the explicit dispatching in basic event loops, it induces an additional complexity with regard to the program structure.

For example, when we implement a drawing application that enables the program to draw objects on a screen, "mouse button pressed" and "mouse button moved" events are delegated to a handler (or listener) that decides, based on the type of the event and its current state, what actions to take. Typically we may distinguish between two states, a state where no object is being drawn and a state where the object is being drawn, the transitions be-

tween the states being triggered with the clicks on the mouse. Dependent on the state the listener may sometimes ignore events. For example button moves are only meaningful when an object is being drawn.

Even in a listener object as simple as the drawing object sketched above, an explicit branching on both object state and event type is necessary. This seems to be inherent in an approach based on listeners.

In server applications it is evident that we may wish to introduce multi-threading to serve multiple clients. However, the need for multi-threading may also occur on the client side, for example when the client must access a remote database: since the network connections may be unreliable, we do not wish to block the entire application while waiting for an answer. With listeners, multi-threading must be introduced in an ad hoc fashion, by creating threads when reacting to an event.

In the following, we will propose an approach based on active objects, which supports multi-threading in a natural and robust way. Our approach, based on an extension of Java, described in the next section, also reduces the need for explicit branching on state and event type, since the invocation of listeners does not need the inversion of control, thanks to the semantics of rendezvous communication.

III. SYNCHRONOUS OBJECT CALLS

The solution presented in this paper rests on a concept of synchronous calls developed at the EPFL [2], [3]. These calls can be introduced into any object oriented language. They could be treated by hand, but we have extended a C++ and a Java compiler to handle them in a controlled manner.

A. general case

Synchronous calls are executed between active objects, i.e., objects that have a distinguished method *run* executed on a thread attached to each object. Let us consider the two classes defined below. It is assumed that the objects instantiated from these classes are provided with threads on which their methods *run* can be executed :

```

class A {
    public void run () {
        for (;;) {
            x = objB.meth();
            System.out.println(x);
        }
    }
}
class B {
    String s;
    public String meth() { return s; }
    public void run () {
        for (;;) {
            // produce a new string in s
            accept meth;
        }
    }
}
A objA = new A();
B objB = new B();

```

The call *objB.meth()* from *objA* (1) is blocked as long as *objB* has not executed the statement *accept meth* (2). *objB* is also blocked on its *accept* statement if *objA* is not calling at the time *objB* arrives at its *accept*. The call from *objA* to *objB* creates thus a rendezvous between these two objects, which is a standard communication means, like the one defined by CCS [4] for example. The implementation of this synchronization relies on a kernel that blocks its users until it finds a matching call-accept pair.

Blocking calls are natural: when a method returns, the data are available and/or the operation is finished. However, they require a way to handle non-determinist event occurrences. Our proposal introduces the *select* statement explained in the following for that purpose.

```

class A {
    public void myMeth() {...}
    public void run () {
        for (;;) {
            select {
                case
                    x = textField.getText();
                    System.out.println(x);
                case
                    button.pressed();
                    System.out.println("pressed");
                case
                    accept myMeth;
            }
        }
    }
}

```

```

}
}
}

```

Note that this statement is different from Ada's *select*: unlike Ada's, it may contain both calls and *accepts*. The meaning of our statement is the following. When the program arrives at the *select* statement, it communicates to the kernel the *calls* and the *accepts* it is ready to execute and it waits for the kernel to find a complementary *call* or *accept*. As soon as one is available, the number of the case selected by the kernel is returned to the program and the program executes that case.

B. Application to events

The approach described above can easily be extended to handle the communications between threads and listeners, as illustrated by the following example.

```

class SButton extends Jbutton {
    public void pressed() {}; // blocking
    addActionListener(new ActionListener() {
        public void actionPerformed
            (ActionEvent e) {
            accept pressed;
        }
    });
}

```

Class *SButton* is a *JButton*, but it contains a blocking method named *pressed*. This method has no parameters nor code, one just uses its name to identify the synchronization. A call to method *pressed* is blocked until the button is clicked on the window and the statement *accept pressed* is executed in the listener (which needs some extra code included by the compiler or by hand). This *accept* must be handled in a particular way, because it is called by the GUI thread and may thus not be blocking. The kernel just registers the fact that method *pressed* is ready to be executed, without blocking the thread that accepts the method. On the other hand, the call to *pressed* from the user thread is blocked exactly in the same way as with an *accept* located on a user thread. *SButton* could be stored in a library and used in all applications without any modification.

Using such a set of synchronous buttons is straightforward:

```
class User {

    SButton buttonOK = new SButton("OK"),
            buttonNOK = new SButton("NOK");

    public void run () {
        for (;;) {
            // display a question
            select {
            case
                buttonOK.pressed();
                // handle OK response
            case
                buttonNOK.pressed();
                // handle Not OK response
            }
        }
    }
}
```

This program is very clear, because the FSM (finite state machine) it represents can be extracted easily. It cannot be called by any event that it is not ready to handle. The developer need not be aware of the concept of listeners or events to write or understand the program. Except for the statements that place the buttons in a window, the source code above is complete and ready to be compiled with a compiler that understands the *select* statement. Hand coding that statement is simple, even simpler than what is proposed in JCSP (Java CSP, see section VI and [5]).

IV. EXAMPLE: NON-BLOCKING GUI APPLICATION

This section describes an example of a program that needs to handle several events in a particular order, and compares the implementation with listeners and with the concept of synchronous calls described above.

The purpose of the program is to wait for a button click that indicates which element the user wants to draw and then to draw a rubber band on a particular drawing area in order to get the two end points of a line segment, respectively the points defining an arc.

A. Version based on listeners

We first show the construction of a standard program. According to what is proposed in the frame of Swing, for example, the method that handles the events is registered in a listener attached to the button and in another listener attached to the mouse object. It is thus called by all events produced by the clicks on the mouse whenever the cursor is in the drawing area. Depending on what has already occurred, different events must be handled, in various manner, or ignored. One could add and remove the listeners during the execution to have only the expected events call the method, but that would be very complicated. It is much simpler to ignore the unexpected events.

Here is a possible version of the source code.

```
public void run(AWTEvent mev){
    switch (state) {
    case start: // program start
        if (mev.getSource()==segmentButton){
            grArea.setXORMode();
            state = seg1;
        } else
            if (mev.getSource()==arcButton){
                grArea.setXORMode();
                state = arc1;
            }
        return;
    case seg1: // get the first point
        if ((mev.getSource()==grAreaMouse)
            &(mev.getID()
                ==MouseEvent.MOUSE_PRESSED)){
            p1 = ((MouseEvent)mev).getPoint();
            p3 = p1;
            state = seg2;
        }
        return;
    case seg2: // handle the rubber band
        if (mev.getSource()==grAreaMouse)
            if (mev.getID()
                ==MouseEvent.MOUSE_DRAGGED){
                // erase previous line
                grArea.drawLine(p1,p3);
                // draw new position
                grArea.drawLine(p1,p2);
                p3=p2;
            } else if (mev.getID()
                ==MouseEvent.MOUSE_RELEASED)
                state = seg3;
                . . .
    }
```

Variable *state* memorizes the next point of

execution. *segmentButton* is the button that triggers the drawing of a segment, *grAreaMouse* represents the source of the mouse events and *grArea*, the graphical area where the drawing must be done.

In the *start* state, the method waits for an event either from the button that asks the editor to draw a segment or from the one that requires an arc. When the first of these events has occurred, the XOR graphical mode is set to draw the rubber band, variable *state* is set to the next state and the method returns to the system. The next time it is called, the method jumps to case *seg1*, where it checks if the incoming event indicates that the mouse has been pressed. In that state, it ignores the other events, such as *MOUSE_MOVED*. When event *MOUSE_PRESSED* occurs, the point where the cursor lies is read from the event, and variable *state* is set to the next state, which waits for the occurrence of the *MOUSE_DRAGGED* event. And so on.

If the operations defined in other objects also require the reading of the mouse events, they can register their listeners too. All objects just have more events to ignore.

Notice that the *switch* statement presented above is used to provide a systematic way to split a program at all locations where an event must be waited for.

B. Solution based on rendezvous

The solution based on rendezvous makes it possible to store the code that draws a segment and the one that draws an arc in different methods which clarifies much the code source. The first part below just decides if the user wants to draw a segment or an arc.

```
public void run() {
    for (;;) {
        select {
            case
                segmentButton.pressed();
                // call the corresponding method
                drawMySegment();
            case
                arcButton.pressed();
                drawMyArc();
        }
    }
}
```

Method *drawMySegment* listed below shows how to draw the rubber band and then the definitive line. It runs on the thread carrying out method *run* above. *grAreaMouse.pressed()* and *grAreaMouse.dragged()* are assumed to be blocking methods that are executed only when the mouse button is pressed or dragged.

```
public void drawMySegment() {
    boolean moving;
    grArea.setXORMode(Color.white);
    p1 = grAreaMouse.pressed();
    p3 = p1; // initialize p3
            // handle rubber band
    for (moving=true;moving;) {
        select {
            case
                p2 = grMouse.dragged();
                grArea.drawLine(p1,p3);
                grArea.drawLine(p1,p2);
                p3 = p2; // remember last point
            case
                p2 = .grMouse.released();
                grArea.drawLine(p1,p3);
                grArea.setPaintMode();
                grArea.drawLine(p1,p2);
                moving = false;
        }
    }
}
```

Replacing that code by a code that uses listeners directly could be done systematically, but is not an easy task if it must be done without a tool.

V. SOFTWARE ENGINEERING PERSPECTIVES

Non-determinism is an essential ingredient of event-based system. From a software engineering perspective, the problem is not to suppress this non-determinism, but to develop modular and robust code, which may easily be maintained and extended when the application requirements change.

Explicit switch statements are rightfully called a *maintenance nightmare*, and are consequently made obsolete by the inheritance mechanism that is characteristic of object-oriented technology, [6]. As we have argued in section II, listener objects may be regarded as another way to suppress explicit dispatching. However, there is a price to pay, namely

we need to accept an *inversion of control*. Although listeners make a modular approach possible, i.e., an indefinite number of listeners may be created and registered to deal with events, it is not always easy to get insight in the flow of control, i.e., to know in which order the events activate the various parts of the program. This problem is aggravated by the need to do explicit branching both on the state of the listener object and on the type of the event.

Our approach, described in the previous sections, actually obviates the *inversion of control*, owing to the semantics of active objects and synchronous communication by rendezvous. As a consequence, it becomes easier to analyze the flow of control and the patterns of activity displayed by the program in response to events.

Figures 1 and 2 allow us to highlight the improvement that can be expected from our approach. In figure 1, a program state is represented by a set of elementary states of the GUI elements and a transition is defined by the enable/disable actions executed within the methods called by the listeners. However, there are many means for doing this, and these enable/disable actions can be stored in any order in the source code, making it very difficult for a developer to analyze them, and impossible for a tool to recover the FSM that specifies the behavior of the application by reverse engineering. On the other hand, in figure 2, the program states are clearly defined at all its locations. So are the transitions between the

states.

Synchronous calls provide an additional benefit: they make it very easy to react to the network events (termination of remote method invocation or arrival of a message). In figure 2, the program has two pending calls in the last state: the left transition waits for a cancel button and the right one checks if the network access is terminated. If the network is congested, the user can click the button and the program can then call a method that aborts the network transmission, without relying to new threads or extra concurrency gimmicks.

VI. RELATED WORK

Our approach is related to JCSP, a Java extension based on CSP (Concurrent Sequential Processes). However, JSP uses channels for communication by rendezvous, whereas we use synchronous method calls. Channels require explicit data coding, hence our approach fits in more naturally with an object paradigm, where communication between objects is statically characterized by method interfaces.

Our *select* statement is close to Ada's one. However, Ada does not allow calls in its *select* statements, which makes it impossible to handle the GUI without callbacks. An Ada program can determine when it accepts the calls, which is already an improvement, but the GUI must be told which entry must be called by each event, an indication that may change while the program is running.

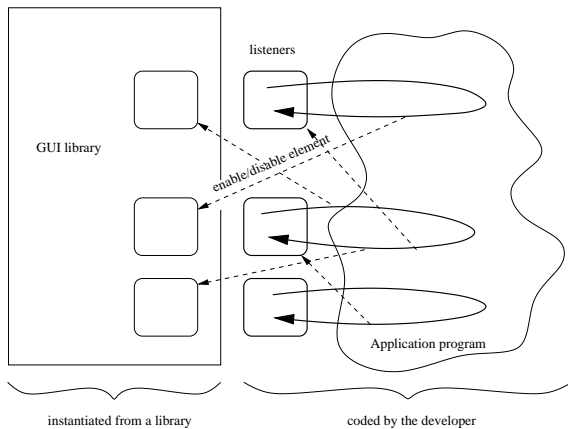


Fig. 1. Program structure with listeners

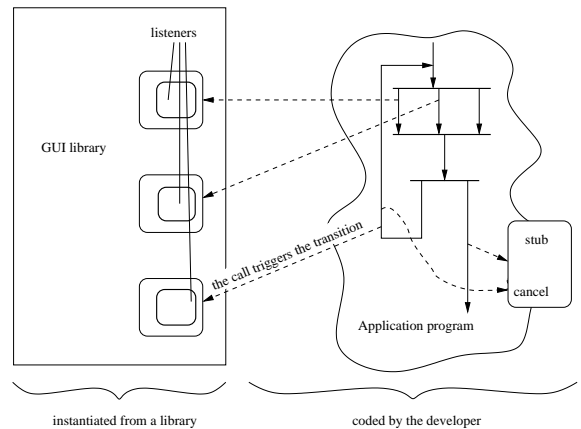


Fig. 2. Program structure with rendezvous

Our proposal of active objects should also be compared with the concept of Actors [7]. An Actor is an object that may acquire threads. Instead of blocking a method in the way we propose, the caller of an Actor's method deposits the parameters of its call (method name and parameters) in a queue and the Actor's thread gets the calls one after the other, when it has the time to do it. The caller is not blocked at once, but only when it uses the result returned by the method (wait by necessities). This approach makes it very difficult to identify program states and it is still not possible to read two buttons in parallel. Actually, that would require a possibility to read either one of two variables, depending on which one gets its content first.

VII. CONCLUSION

This paper has presented an approach to event-handling based on active objects and synchronous communication. This approach is particularly well suited to the development of interactive applications relying on graphical user interfaces and network communications. In the example, we have shown that our approach allows a more direct way of intercepting events, which leads to a significant simplification of the code. Also the *flow of control*

becomes easier to understand, since the use of synchronous communication obviates the need for an *inversion of control* as is characteristic for GUI libraries using listener objects to deal with events.

ACKNOWLEDGMENTS

It is our pleasure to acknowledge the help of Matthias Zenger and David Cavin for the extension of the Java compiler with synchronous calls and of Vo Duc Duy for the development of a library of synchronous GUI elements.

REFERENCES

- [1] D.C. Schmidt, *Experience using design patterns to develop reusable object-oriented communication software*, CACM 38(10), pp 65-74, 1995.
- [2] C. Petitpierre, *Synchronous C++, a Language for Interactive Applications*, IEEE Computer, September 1998, pp 65-72.
- [3] <http://ltiwww.epfl.ch/sJava>
- [4] R. Milner, *Communication and Concurrency*, Prentice Hall, Englewood Cliffs, N.J., 1989.
- [5] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 2000.
- [6] A. Eliëns, *Principles of Object-Oriented Software Development*, Addison-Wesley, 2000.
- [7] G. Agha, *A Model of Concurrent Computation in Distributed System*, The MIT Press, 1986.
- [8] E. Gamma et al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley.