title: Event-handling with active objects – dealing with non-determinism in high-performance applications

fragments (in construction)

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 listeners provided by the application. In this paper we will argue that event-handling with (passive) listener objects lead to problems in high-performance applications, which require non-blocking actions on the occurrence of events. Active objects provide a uniform approach to robust event-handling, both with respect to the registration and (de)activation of listener objects, as well as the (non-blocking) dispatching of events. We will illustrate our approach by discussing how to develop a graphical user interface for anapplication that must access a remote database. We will also indicate that as a benefit, from a software engineering perspective, our approach allows for a more modular approach to event-based systems and better insight in the flow of control between listeners and the application.

1 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 multithreading is needed, for example to service multiple clients or preventing the application to block when accessing a remote resource.

In this paper we propose an approach to event-handling based on active object and synchronous rendez-vous. The structure of the paper is as follows. In section 2, 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 multithreading. In section 3, we will provide a brief description of sJava, an extension of Java that supports active objects and synchronous communication by rendez-vous. In section 4, we will proved an example that illustrates how synchronousn communication allows for a more direct approach to event-handling. In section 5, we will discuss the benefits of our approach from a software engineering perspective. And in section 6, we will draw some conclusions.

related work Our approach is related to JCSP, a Java extension based on CSP (Concurrent Sequential Processes). However, JSP uses channels for communi-

cation by rendez-vous, whereas sJava uses 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.

2 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 an environment that waits for events to occur. Handler function or handler objects are then invoked for an appropriate response. 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 the 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 allows for extensions since new callback functions may 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 inbetween (hook) method invocations, listener objects are strictly more powerful then callback functions that must rely on ad hoc mechanisms to take the history of event occurrences into account.

The *Reactor* pattern, described in [Schmidt95], 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*, that is the application code must await until it is invoked by the (event) dispatching mechanism. Although this induces an additional complexity with regard to the program structure, it is definitely an improvement over the explicit dispatching in basic event loops.

For example, when we implement a drawing application that enables us to draw objects on a screen, button press and move events are delegated to a handler (or listener) that decides based on the type of the event and its state what actions