Events Not Equal To GUIs*

Stuart Hansen
Department of Computer Science
University of Wisconsin - Parkside
Kenosha, Wisconsin
hansen@cs.uwp.edu

Timothy Fossum
Department of Computer Science
University of Wisconsin - Parkside
Kenosha, Wisconsin
fossum@cs.uwp.edu

ABSTRACT

The event driven paradigm is ubiquitous in modern software. Many texts introduce events when discussing graphical user interfaces, but the event paradigm extends well beyond that domain. Events also play important roles in operating systems, component based systems, reactive systems, middleware, web services and other fields. Computer science educators have an obligation to see that our students thoroughly understand the event paradigm and have some grounding in tools to develop event driven systems. This paper describes an upper division, computer science elective course in event driven programming. The course gives a comprehensive treatment of event driven systems. It appropriately captures the importance of the event paradigm and serves to integrate concepts from several different computing fields, including Programming Languages, Operating Systems, and Software Engineering. It also introduces students to advanced tools and packages designed for developing event driven systems. The course has been taught four times at our institution, having repeatedly received high marks from the students for both its conceptual and technical content.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.1.m [**Programming Techniques**]: Miscellaneous

General Terms

Design, Languages

Keywords

Events, Event Driven Programming

1. INTRODUCTION

Event driven programming is an important conceptual and practical programming paradigm. It is used in and influences many different fields of computing. Often textbooks contain sections or chapters on events as they relate to the text's topic [2, 6, 7, 10, 23]. The 2001 computer science curriculum guidelines include event driven programming as a topic within programming fundamentals [8]. Numerous computer science educators have promoted the use of event programming in the undergraduate curriculum [4, 20, 21, 22]. The emphasis by these authors tends to be to introduce events and GUIs early in the CS curriculum.

Event driven programming is important enough to warrant its own course, in addition to treatments elsewhere in the curriculum. We have introduced a senior level elective on event driven programming. The course treats event driven programming as a comprehensive paradigm that touches on many different application areas. While GUI development serves as an important source of examples, it is only one of several types of systems discussed in our course. Instead, the course looks at the common fundamental ideas behind a variety of event driven systems and the tools to develop them. We present these ideas as a unifying framework on which to ground our studies.

We believe that a comprehensive and unifying treatment of event driven systems appropriately captures the importance of the event paradigm and serves as an opportunity to put these concepts on a theoretical footing, much as a course on programming languages does after a student has had exposure to introductory programming in specific languages. The ubiquitous presence of events in modern computing systems makes it easy to justify their treatment in the CS curriculum. Modern object–oriented languages like Java and C# include complex class hierarchies to handle events. Integrated development environments (IDEs) for these languages often have special built–in functionality to facilitate event programming. At a much lower level, interrupt processing also fits into our paradigm. For example, a physical mouse click is captured and processed by the operating system via interrupts. In fact, the passing of a mouse click through the various layers, from hardware to operating system, to windowing system, to application, makes an excellent classroom example.

2. THE EVENT PARADIGM

Early in our course we present a comprehensive conceptual model of event driven programming. Many different programming languages support the development of event driven systems [1, 3, 5, 9, 11, 14, 15, 17, 18, 19]. The programming model put forward by each of these varies and can be a matter of considerable debate [15, 18], which makes defining a comprehensive model more difficult. Our approach is to distill the features that are common to many of these systems.

*Supported in part by NSF grant DUE #0089406

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

2.1 Characteristics of Event Driven Programs

We have identified several features that underlie event driven programming systems. Principal among them are loose coupling, state based control and concurrent processing.

2.1.1 Loose Coupling

In object–oriented systems, events have a source object and a handler object. Events originate in the source and are processed by the handler. Event Sources may be any object, including GUI (or non-GUI) components, hardware devices or remote processes. In each case the event source fires an event to the handler, which responds by taking some action. The same source/handler relationship exists in non–object–oriented systems. Rather than having a handler object, however, the handler is typically a callback function.

The relationship between sources and handlers generally has the following characteristics:

[1] Runtime Registration

The binding of event sources and handlers is delayed until runtime. In procedural languages, callback functions are registered with an event source using function pointers. In object–oriented languages like Java, the event handler objects are registered with the event sources by passing a reference to the handler to a registration method in the source.

While event registration (and deregistration) is a runtime activitiy, good event driven designs consistently register handlers as one of the first steps of program execution and never deregister them. Early registration is so fundamental that some event driven systems, like Java Beans, give registration time a special name, "Design Time" [9]. If the system designer needs to change the response to an event, the designer can use the state design pattern (discussed below) to alter the behavior of the handler rather than registering an alternative handler.

[2] Multicasting

Multicasting means that a single event may be sent to multiple event handlers. A typical example would be when an object has multiple views associated with it. Each view needs to be updated when a property is changed, so each is sent the event.

[3] Multiplexing

Multiplexing occurs when a handler receives events from multiple sources. GUIs use multiplexing when there are multiple ways to accomplish the same action, e.g. choosing File – Save or clicking on a Save icon.

[4] Inverted Semantics

In procedure–oriented design, programs are often implemented in units that have a client/server relationship. Client procedures achieve their higher–level goals in part by calling a collection of procedures that provide lower–level services. In such cases, the client blocks until the server completes. Frequently the server procedures return values to the clients either through return values or indirectly through side–effects. The client can be viewed as having a "higher purpose" than the server procedures it may call.

In many event driven programs, a low–level event source triggers a higher–level action in a handler, e.g. a mouse click may trigger a handler to delete a file. The event source never receives a return value and (because of late binding) will not know what actions are triggered by the registered handlers. Often the event source does not block to wait for handlers to complete their tasks.

2.1.2 State Based Control

An event driven system consists of a collection of objects that change as events propagate among them. In simple systems the objects' data may be the only thing that changes. For example, an event results in a name being added to a class roster, or the price of popcorn being increased by a quarter. In more complex systems, the control state of the system also changes. The control state determines what events the system responds to, and how it will respond to each of them. At any time, the system is in a specific control state. For example, a stopwatch is in one of two states: stopped or running. Only if the state is running does the time get updated; when stopped, the watch ignores the advance of time. The control state of the watch determines how it responds to timer events. Many real world embedded systems use state based control, but very few of our students have been exposed to the design of a state based system before taking our course. We spend a significant amount of time discussing state based control, the state design pattern [13], and UML models for representing control state [12, 16].

2.1.3 Concurrent and Distributed Computing

Multithreading and concurrency are intimately related to event driven programming. These topics are generally introduced to undergraduate students in an operating systems course. Frequently students study the standard synchronization problems, but are not asked to do significant concurrent programming. Our event driven programming course gives them a chance to integrate their understanding from operating systems while solving practical programming problems. The classic synchronization issues of race conditions and deadlock are faced by students even when developing fairly simple event driven systems.

In distributed event driven systems the event sources and handlers are spread across multiple computers. An event handler receives a message from a possibly distant source telling it that an event has occurred. The event handler processes the event in the same way it would process local events. The current generation of college students has grown up with the Internet. They are very familiar with chat servers and multi–user distributed games. Our course gives them a first look at the technical issues related to designing and developing distributed systems.

3. COURSE OUTLINE

This section presents a brief overview of our course. We faced a basic problem of choosing what development tools and systems to use. Many different languages and tools support event driven programming. We felt that the best approach was to give a more in–depth treatment of a few of them rather than to give a cursory treatment of many. Our two primary goals in choosing tools were that the tools be relatively popular and that together they give the students a good breadth of experience.

3.1 GUIs

We begin the semester by teaching Java GUI development using Swing. In CS1 and CS2, we introduce students to Java GUIs, but only briefly. Our treatment of GUIs in the event driven programming course is more extensive. Students explore many of the standard Swing components. They gain technical knowledge in Swing programming and an appreciation for the complexity of human computer interfaces.

During this unit students also learn the Java event model. We introduce the concepts of event sources, handlers, listeners and adapters and explain the model–view–controller paradigm.

Toward the end of our GUI unit students study the Java implementation of events and components. We assign them a project where they define their own event types, develop a source component for those events, define a listener interface for the event class and supply a handler that implements the interface. This challenging project integrates many concepts. Students gain a deeper understanding and appreciation of the roles the various classes and interfaces play.

3.2 Concurrent Event Programming

We introduce concurrent programming to our students when discussing the event monitoring thread. This thread bears the responsibility for monitoring all event occurrences. It detects the events and hands them off to the appropriate handlers. This thread needs to remain responsive. It should detect events very shortly after they occur. If an event handler's processing will take much time, it must execute in a new thread, leaving the monitoring thread to wait for more events. Thus, the event monitoring thread may detect multiple events and may start multiple handler threads in quick succession. Since individual handlers may have multiple threads executing concurrently within them, this provides an opportunity for us to discuss synchronization issues.

3.3 Distributed Event Programming

The loose coupling between event sources and handlers makes event driven programming a natural domain for distributed processing, since the event source and event handler may be on separate computing systems. There are many libraries and run-time systems for building and deploying event driven distributed applications. We chose CORBA and web services as example distributed architectures that support such activities.

CORBA, the Common Object Request Broker Architecture, is a middleware system designed to help integrate legacy systems into a distributed environment. There are CORBA implementations available for a wide variety of hardware platforms, operating systems and languages. This has made CORBA a popular system for building distributed applications.

CORBA places an object request broker (ORB) behind each component of the system. At run-time, the ORBs communicate with each other using protocols that are machine- and language-independent. A programmer lets CORBA auto–generate most of the code needed for the application objects to work with their ORBs. The distributed nature of the system is almost transparent to the application programmer. The event source calls an auto–generated local proxy event handler. The proxy works with the ORB to deliver the event to the actual handler on the remote system.

One of our more successful assignments required students to develop a chat system, including both a server and a client. Every student's client was able to communicate with every other client using any of the server implementations. A student's code typically involved 20 classes of their own, plus approximately 15 classes auto–generated by CORBA. Hundreds of classes and objects worked together seamlessly in the resulting distributed system. This is a testament to the quality of Java's CORBA implementation and how easy it is for students to learn distributed processing using CORBA.

We also introduce students to web services. Many CS departments are looking for ways to integrate web services into their curricula. We found it to be a natural match with our event driven programming course. A web service runs behind a web server and uses XML (eXtensible Markup Language) to communicate with its clients. Requests for the program's services arrive at the server in the form of XML. The service parses the XML, executes the request and returns the result, again in XML. Clients may be web browsers, standalone applications, or other web services. The web server gives a safe environment for service execution, and XML gives the flexibility to deal with diverse clients.

In our early offerings of our course students developed web services and clients in Java. Microsoft's new Visual Studio .NET provides a powerful, easy to use platform for this type of development. In our most recent offering, students developed their web services in .NET.

3.4 Engineering Event Driven Systems

As a final project, students develop a larger event driven program of their own choosing. We discuss some of these projects in the next section. While the students are working on their projects, class time is dedicated to an exploration of the software engineering aspects of event driven programming. We emphasize those aspects of software engineering that are particularly applicable to the event driven programming paradigm.

There are several UML models that are useful in event driven systems. Interaction diagrams capture the expected sequence of events. Activity diagrams illustrate the concurrency and synchronization of a system; and state charts represent the control states of the system. We also introduce more traditional models of concurrency such as Petri Nets.

Testing and debugging event driven systems pose challenges not present in many other types of development. We emphasize state based testing in our discussions. Each possible event should be tested adequately in each of the possible control states. The loose coupling between sources and handlers makes debugging event driven systems difficult. An event may not result in an expected behavior if the event did not fire, if the handler was not registered, or if there is a bug in the handler.

Discussing software engineering while students are working on projects has proved particularly effective. Students find the UML models useful in their project designs. The projects are sophisticated enough that the students appreciate a disciplined approach to testing and debugging their systems.

3.5 Other Topics

The event driven programming paradigm extends well beyond the topics outlined above. When time permits we also discuss the following topics in our course:

3.5.1 Low Level Event Processing

Interrupt handling is event driven. Students gain a deeper understanding of both operating systems and the event driven paradigm when interrupts are presented in this manner.

3.5.2 Simulation

Event driven programming evolved in part from discrete event simulation. Discrete event simulation is large topic. We have introduced students to it in one or two lectures, but have not developed a complete unit on it.

3.5.3 Games

Students love games and game programming. Complex tool kits exist to help programmers develop games, and most of these toolkits contain support for events. While some of our students choose to work on projects that are game-related, sophisticated game programming extends well beyond the scope of our course, and we

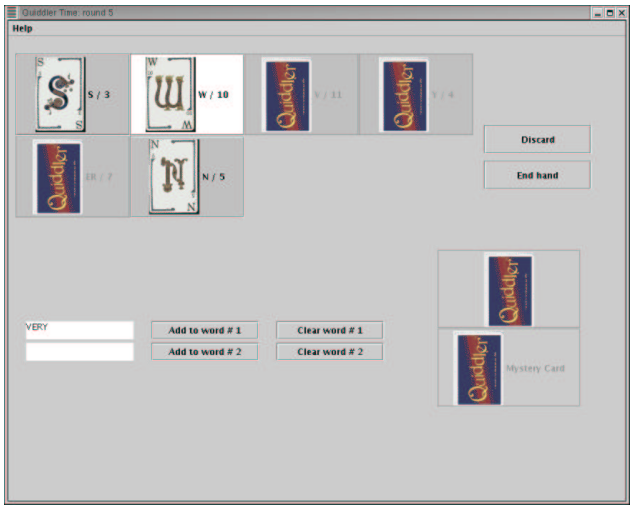

Figure 1: The GUI for the Quiddler client program. This client has been used to play Quiddler across widely geographically dispersed machines.

give it little more than a cursory treatment.

4. STUDENT PROJECTS

One of the real pleasures of teaching event driven programming has been the eagerness and energy students have demonstrated when it came time to develop their own projects. The instructors have guided the projects primarily by helping the students establish a reasonable scope, but it has been the students who have come up with a broad array of original project ideas. On multiple occasions students have even chosen to explore and use event driven technologies that we had not discussed in class. This section briefly discusses a few of the diverse projects that students have completed.

4.1 GPS Mapping System

One student developed a system to connect his handheld Geographic Positioning System (GPS) with his laptop. The laptop maps the route followed as the student drives or walks around. The implementation uses the Java Communications API which is event driven. This student spends his summers working on search and rescue teams in the Rocky Mountains. He hopes to extend the system to include topological map images on the background of his screen.

4.2 Quiddler®

Another student developed an especially nice distributed Internet implementation of Quiddler[1], a card game where players score points by forming words. The user interface for the Quiddler client is shown in Figure 1.

4.3 An Event Monitor

A team of two advanced students developed an event monitoring system used as a tool to help programmers see what events are processed by their programs. The system captures the Java events that occur within a program and then displays them in meaningful ways. The system was developed in AspectJ, an aspect–oriented language that produces code that instruments a running Java program without changing the Java program's semantics. A separate window displays the events as they occur.

[1]Quiddler is a Registered Trademark of Set Enterprises Inc.

5. CONCLUSIONS

Our Event Driven Programming Course has been a major success. It helps students gain an in depth understanding of the event paradigm and it exposes students to modern programming tools designed for event driven programming. Our students have found the course material to be important to them both theoretically and practically.

6. REFERENCES

[1] E. Angel. *Computer Graphics: A top-down approach with OpenGL*. Addison-Wesley, 1997.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1968.

[3] G. Brose, A. Vogel, and K. Duddy. *Java Programming with CORBA*. OMG Press, 2001.

[4] H. B. Christensen and M. E. Casperson. Frameworks in cs1 - a different way of introducing event-driven programming. *SIGCSE Bulletin*, 34(3):75–79, September 2002.

[5] F. Culwin. *A Java GUI Programmer's Guide*. Prentice-Hall, 1998.

[6] H. M. Deitel and P. J. Deitel. *Java: how to Program, 4th Edition*. Prentice-Hall, 2002.

[7] H. M. Deitel, P. J. Deitel, J. Listfield, T. Nieto, C. Yaeger, and M. Zlatkina. *C#: how to Program, 4th Edition*. Prentice-Hall, 2002.

[8] G. Engel and E. Roberts, editors. *Computing Curricula 2001 – Computer Science*. ACM and IEEE, 2001. Online. Internet. Available WWW: http://www.acm.org/sigcse.cc2001.

[9] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[10] E. C. Epp. *Prelude to Patterns in Computer Science using Java*. Franklin, Beedle and Associates, 2001.

[11] A. R. Feuer. *MFC Programming*. Addison-Wesley, 1997.

[12] M. Fowler. *UML Distilled, 2nd ed.* Addison-Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2000.

[14] M. Henning and S. Vinoski. *Advanced CORBA Programming*. Addison-Wesley Longman, 1999.

[15] Java Language Team. About microsoft's "delegates", 2000.

[16] C. Larman. *Applying UML and Patterns: An Introduction to Object–Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 2 edition, 2002.

[17] S. B. Lippman. *C#: A Practical Approach*. Addison-Wesley, 2002.

[18] Microsoft Corporation. The truth about delegates, microsoft white paper, September 1998.

[19] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, 2002.

[20] J. Raab, R. Rasala, and V. K. Proulx. Pedagogical power tools for teaching java. *SIGCSE Bulletin*, 32:156–159, 2000.

[21] R. Rasala, J. Raab, and V. K. Proulx. Java power tools: Model software for teaching object-oriented design. *SIGCSE Bulletin*, 33:297–301, 2001.

[22] L. A. Stein. What we've swept under the rug: Radically rethinking cs1. *Computer Science Education*, 8:118–129, 1998.

[23] A. Tucker and R. Noonan. *Programming Languages: Principles and Paradigms*. McGraw-Hill, 2002.