Event-Driven Programming: Introduction, Tutorial, History

http://Tutorial_EventDrivenProgramming.sourceforge.net

Stephen Ferg (steve@ferg.org)

Version 0.2 – 2006-02-08

Revision History

(most recent first)

?	version 1.0 released
?	version 0.3 – Modified beta version
2006-02-08	version 0.2 – modified draft version
2006-01-10	version 0.1 – First draft

Acknowledgments

Thanks to my co-worker Jeff Davis, for scanning several of the diagrams.

Thanks to my co-worker Gary Ault, who reported a mistake in one of the state transition diagrams.

Contents

Introduction

It's not easy to learn event-driven programming. If you're trying to program your first GUI application, or trying to learn how to parse XML with a SAX parser, you've experienced the difficulties first-hand.

> *Most, if not all, GUI systems and toolkits are designed to be event driven, meaning that the main flow of your program is not sequential from beginning to end. If you've never done GUI programming, this is one of the trickiest paradigm shifts.*
> — Robin Dunn, speaking on GUI programming at OSCON2004
>
> *Hollywood Principle: "Don't call us; we'll call you." ... You implement the interfaces, you get registered. You get called when the time is right. This requires a distinctly different way of thinking to that which is taught in introductory programming where the student dictates the flow of control.*
> — Dafydd Rees, http://c2.com/cgi/wiki?HollywoodPrinciple

The problem isn't that the concepts are complicated or difficult – the basic ideas are really quite simple. The problem is that – as of January 2005 – no reasonably complete explanation of these concepts is available on the Web.

This paper is my attempt to change that. Like so many authors, I've written the paper that I wish I could have found when I needed it. I hope it will help you in your attempt to learn event-driven programming.

One effective way to explain a complex idea is to tell the story of its life. This kind of story begins with the initial appearance of a young, simple idea. As the story unfolds we watch the progress of the idea as it responds and adapts to changing circumstances, slowly growing in complexity. This kind of story is called a *genetic explanation* and it is what I try to do in the first part of this paper.

This story of the evolution of event-driven programming is told from the perspective of a business applications programmer who started programming in the late 1970's, worked mostly on IBM and Microsoft platforms, and most recently began working with Java and Python on Unix platforms. A professor of computer science – or someone who worked on IBM's CICS transaction processing monitor, or on the Mesa programming environment, or on the Andrew windowing system – would undoubtedly tell a different story, or at least tell it differently. So this is not the only way the story could be told; it is simply the story as I am able to tell it.

The code examples are mostly in pseudo-code and Python, with the occasional bit of Java. Python is such a straightforward language (it's been called "executable pseudocode") that you should be able to understand the Python examples even if you've never seen a line of Python before. For more information about Python, two good places to start are http://www.python.org/ and http://pythonology.org.

In The Beginning – Transaction Analysis

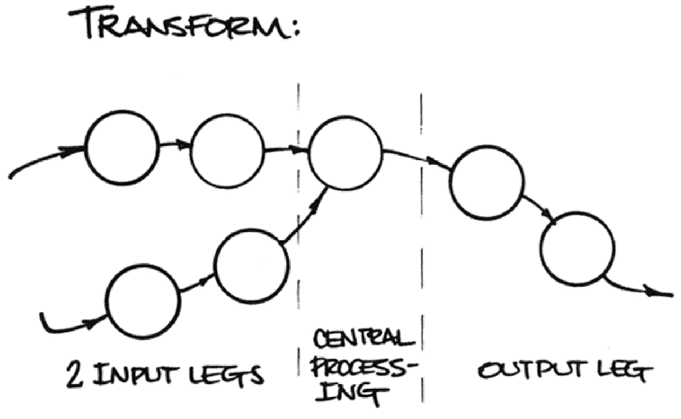
My tale begins in the late 1970's. In those days, the typical computer system was a batch processing system. Input data was typically a sequential file on tape. A program read data from the input file, processed it, and wrote the results out to another sequential file. That file, in turn, was processed by another program and written out to another file, which in turn was processed by another program, and so on. The standard model of a computer system was an assembly line. Raw materials come in one door; they are processed, processed, processed; and at the end of the processing, a finished product is pushed out another door.

This mental model underlay the "structured" systems development methods in the 1970's. The human father of the structured methods was Larry L. Constantine. Their corporate father was IBM's Systems Research Institute. The most successful advocate of structured methods was Edward Yourdon – so much so that the expressions "Yourdon" and "structured analysis and design method" became almost synonymous.

The origin of the term "structured" was a paper called "Structured Design" by Glenford J. Myers, Wayne P. Stevens, and Larry Constantine, that appeared in the *IBM Systems Journal* in 1974. In 1975, Glen Myers, one of Larry Constantine's students at IBM SRI, published *Reliable Software Through Composite Design*. Then in 1977 and 1978, almost simultaneously, several important books on the structured methods appeared – *Structured Design* by Ed Yourdon and Larry Constantine, *Structured Analysis and System Specification* Tom De Marco, *Structured Systems Analysis* by Chris Gane and Trish Sarson, and *Structured Systems Development* by Ken Orr. Perhaps the most influential of these books was Tom De Marco's *Structured Analysis and System Specification*, published through the Yourdon press.

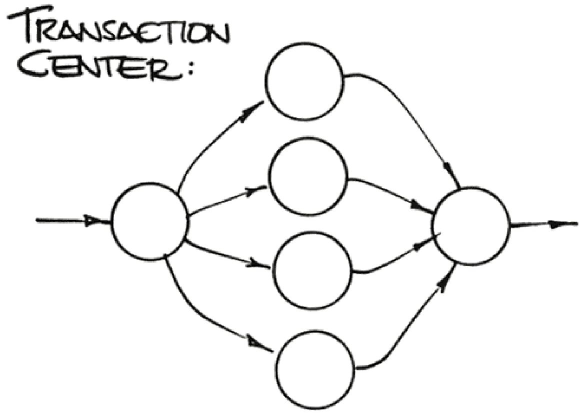
Dataflow Diagrams

Structured analysis used *dataflow diagrams* (DFDs) to show the logical structure of a computer system. On a DFD, a record in a sequential file was conceptualized as a packet of data moving through a pipeline, or along a conveyor belt, called a *dataflow*. Packets passed through a sequence of workstations called *processes* where they were filtered, used, enhanced, or transformed, and then passed on to the next workstation. Here's an example of a dataflow diagram from page 316 of De Marco's *Structured Analysis and System Specification*.



Describing a system in this way was called *transform analysis*.

De Marco also briefly described a second kind of analysis called *transaction analysis* and provided this diagram.



He explained the differences between transform and transaction analysis this way [p. 315]:

> Transform analysis applies to applications that are transforms — that is, applications that have clearly identified input streams, central processing, and output streams. A transform is represented in Data Flow Diagram terms by a linear network.
>
> Transaction analysis applies to transaction centers, parts of the application characterized by sudden parallelism of data flow.

De Marco actually spent very little time discussing transaction analysis. But the topic received more attention in Ed Yourdon and Larry Constantine's book *Structured Design*. In Chapter 11, Yourdon and Constantine give credit for the first description of transaction analysis to one P. Vincent, in a paper called "The System Structure Design Method" published in the limited-edition *Proceedings of the 1968 National Symposium on Modular Programming*. Apparently Mr. Vincent and others at Bell Telephone of Canada had developed a methodology called SAPTAD; Yourdon and Constantine described transaction analysis as "a more flexible, more sophisticated updating of the SAPTAD technique."

"Transaction analysis," Yourdon and Constantine wrote, "is suggested by data flow graphs resembling Fig. 11.1 — that is, where a transform splits an input data stream into several discrete output sub-streams." Here is Figure 11.1. It is the archetype diagram of event-driven programming.

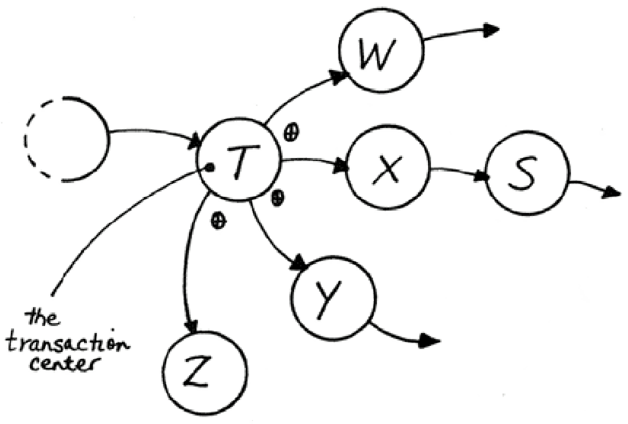


Figure 11.1. Data flow graph of a typical transaction center of an application.

A transaction, they said, begins when "any element of data, control, signal, event, or change of state" is sent to the *transaction center* process.

```
A transaction center of a system must be able to
```

- get (obtain and respond to) transactions in a raw form
- analyze each transaction to determine its type
- dispatch on type of transaction
- complete the processing of each transaction

Structure Charts

A dataflow diagram shows the logical functions that a system must perform, but it doesn't say anything about the design of the program that will perform those functions. In structured analysis and design, a different diagram called a *structure chart* was used to show program design. On structure charts, boxes represent modules (functions, or subroutines). The boxes are arranged hierarchically, with calling modules at the top and called modules beneath them.

Converting a transaction-processing dataflow diagram to a structure chart produced a structure diagram like this one (from *Structured Design*, p. 205).

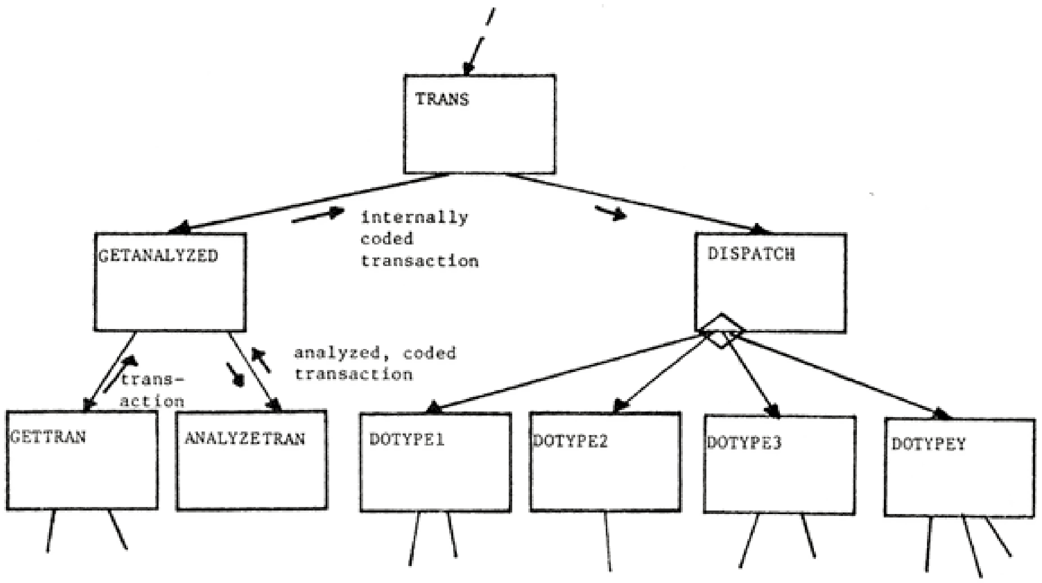


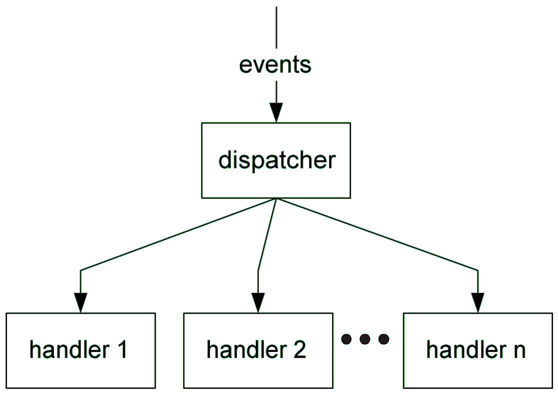
Figure 11.2. Fully factored transaction center.

In this diagram, the dotted arrow coming in from the top represents flow of control being passed to the transaction center. Transactions are obtained by the GETTRAN function. Once obtained, a transaction is analyzed to determine its type (its *transaction code*) and then passed up to the transaction center. From there, it is passed to the DISPATCH module which sends it to the module that handles transactions of that type.

The *Handlers* Design Pattern

If Yourdon and Constantine were writing today, they might very well call their notion of transaction analysis a *design pattern*. I will call it the *Handlers* pattern.

Here's a diagram of the Handlers pattern. The diagram follows the structure of Figure 11.1 – Yourdon and Constantine's original dataflow diagram for transaction analysis.



Handlers pattern

On the diagram you can see:

- a stream of data items called *events* (Yourdon and Constantine's "transactions")
- a *dispatcher* (Yourdon and Constantine's "transaction center")
- a set of *handlers*.

The job of the dispatcher is to take each event that comes to it, analyze the event to determine its *event type*, and then send each event to a handler that can handle events of that type.

The dispatcher must process a stream of input events, so its logic must include an *event loop* so that it can get an event, dispatch it, and then loop back to obtain and process the next event in the input stream.

Some applications (for example, applications that control hardware) may treat the event stream as effectively infinite. But for most event-handling applications the event stream is finite, with an end indicated by some special event — an end-of-file marker, or a press of the ESCAPE key, or a left-click on a CLOSE button in a GUI. In those applications, the dispatcher logic must include a *quit* capability to break out of the event loop when the *end-of-event-stream* event is detected.

In some situations, the dispatcher may determine that it has no appropriate handler for the event. In those situations, it can either discard the event or raise (throw) an exception. GUI applications are typically interested in certain kinds of events (e.g. mouse button clicks) but uninterested in others (e.g. mouse movement events). So in GUI applications, events without handlers are typically discarded. For most other kinds of applications, an unrecognized event constitutes an error in the input stream and the appropriate action is to raise an exception.

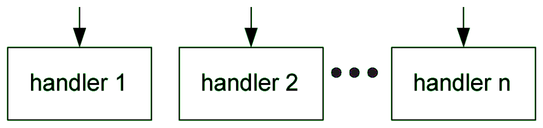
Here is pseudo-code for a typical dispatcher that shows all of these features:

- the event loop,
- the *quit* operation,
- the determination of event type and the selection of an appropriate handler on the basis of that type, and
- the treatment of events without handlers.

```
do forever:    # the event loop

    get an event from the input stream

    if   event type == EndOfEventStream :
         quit # break out of event loop

    if   event type == ... :
         call the appropriate handler subroutine,
         passing it event information as an argument

    elif event type == ... :
         call the appropriate handler subroutine,
         passing it event information as an argument

    else:   # handle an unrecognized type of event
         ignore the event, or raise an exception
```

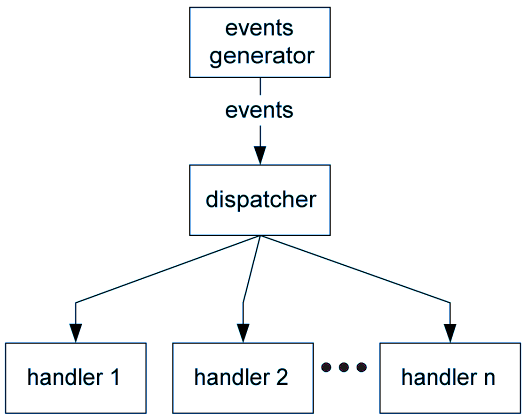
The *Headless Handlers* Pattern

There are a few variants on the Handlers pattern. One of them is the *Headless Handlers* pattern. In this pattern, the dispatcher is either missing or not readily visible. Taking away the dispatcher, all that remains is a collection of event handlers.



The *Extended Handlers* Pattern

Another variant is the *Extended Handlers* pattern. In this variant, the pattern includes an *events generator* component that generates the stream of events that the dispatcher processes.



The Event Queue

In some cases, the dispatcher and its handlers may not be able to handle events as quickly as they arrive. In such cases, the solution is to buffer the input stream of events by introducing an *event queue* into the events stream, between the events generator and the dispatcher. Events are added to the end of the queue as fast as they arrive, and the dispatcher takes them off the front of the queue as fast as it is able.

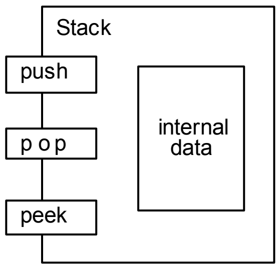
GUI applications typically include an event queue. Significant events such as mouse clicks may require some time to be handled. While that is happening, other events such as mouse-movement events may accumulate in the buffer. When the dispatcher becomes free again, it can rapidly discard the ignorable mouse-movement events and quickly empty the event queue.

Some Examples of the Handlers Pattern

Now that I've described the Handlers pattern, I'd like to show you some examples of the pattern. These methods and technologies are probably familiar, but you may never have thought of them as examples of event-driven programming.

Objects

In the 1990's, object-oriented (OO) technology and methods gradually eclipsed the structured methods of the 1970s and 1980s. Software methodologists began experimenting with new diagramming notations to express object-oriented concepts. At that time, one of the popular diagramming notations (invented by Grady Booch?) was *object diagrams*. Here is an example of an object diagram.



Object diagram of a STACK object

In this object diagram, *Stack* is an object type (or *class*). *Push, pop,* and *peek* are its methods. To use the Stack class, you would create a stack object, and then use its methods to do things to it.

```
# create a stack object by instantiating the Stack class
myStack = new Stack()

myStack.push("abc")
myStack.push("xyz")

print myStack.pop()  # prints "xyz"
print myStack.peek() # prints "abc"
```

I like object diagrams because they clearly show objects as examples of the Headless Handlers pattern. An object diagram is basically the same as the Headless Handlers diagram, except that events arrive from the left rather than the top of the diagram. The Stack class, for example, is clearly a collection of event handlers (called *methods* in OO jargon) for handling *push, pop,* and *peek* events.

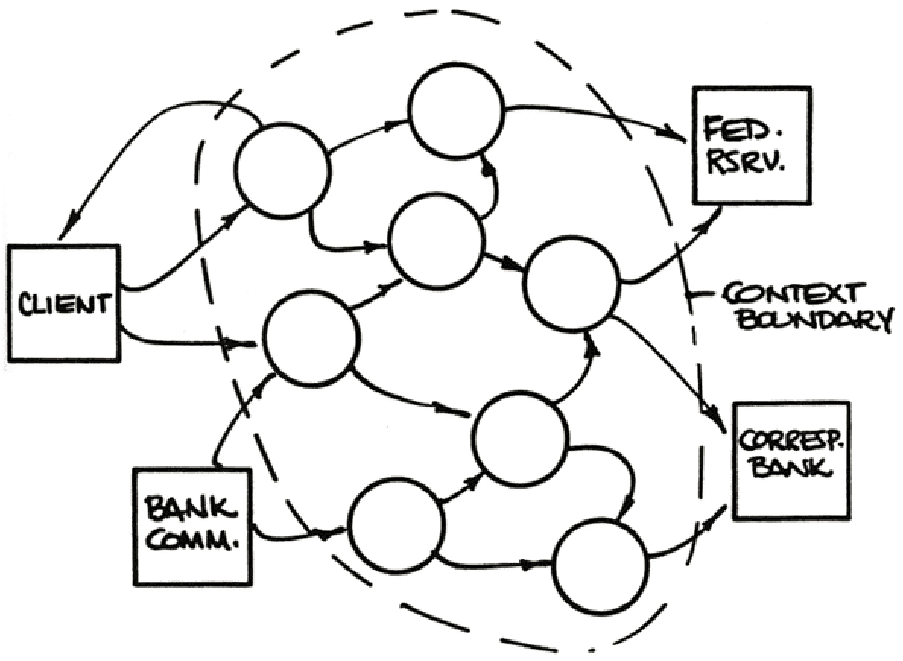
The lesson here is that if you're an object-oriented programmer you *already know* event-driven programming. When you write object methods, you are — quite literally — writing event handlers.

Systems

As we've seen, in Structured Systems Analysis, a computer system was conceptualized as a factory. Raw materials flow into the factory, travel along conveyor belts (data flows) through workstations (processes) and eventually a finished product is pushed out the door.

The suppliers of the raw materials were called *sources,* and the consumers of the finished products were called *sinks*. Sources and sinks were called the *terminators* of data flows – they were the places where data flows began and ended.

A *context diagram* could be used to show the system situated in the context of its terminators. Here is a context diagram from p. 59 of De Marco's *Structured Analysis and System Specification.*



In 1984 Stephen McMenamin and John Palmer published *Essential Systems Analysis*. Essential Systems Analysis (ESA) built on and extended earlier work on structured analysis, but it also introduced a radical change in the conceptual model of computer systems.

ESA regards a computer system not as a factory, but as a stimulus/response machine. The stimuli are events sent into the system by the terminators in the outside world. The system itself is conceptualized as a collection of event handlers (*essential activities*). When an event arrives, the system springs to life, an essential activity processes the event, and then the system goes to sleep again *(quiesces)* until the arrival of the next event.

Essential activities respond to events by reading from and writing to *data stores* at the heart of the system, and by producing output data flows. The system's data stores constitute its *essential memory.*

This diagram (from page 55 of *Essential Systems Analysis*) shows the essential parts of a computer system, as they were conceptualized in ESA.

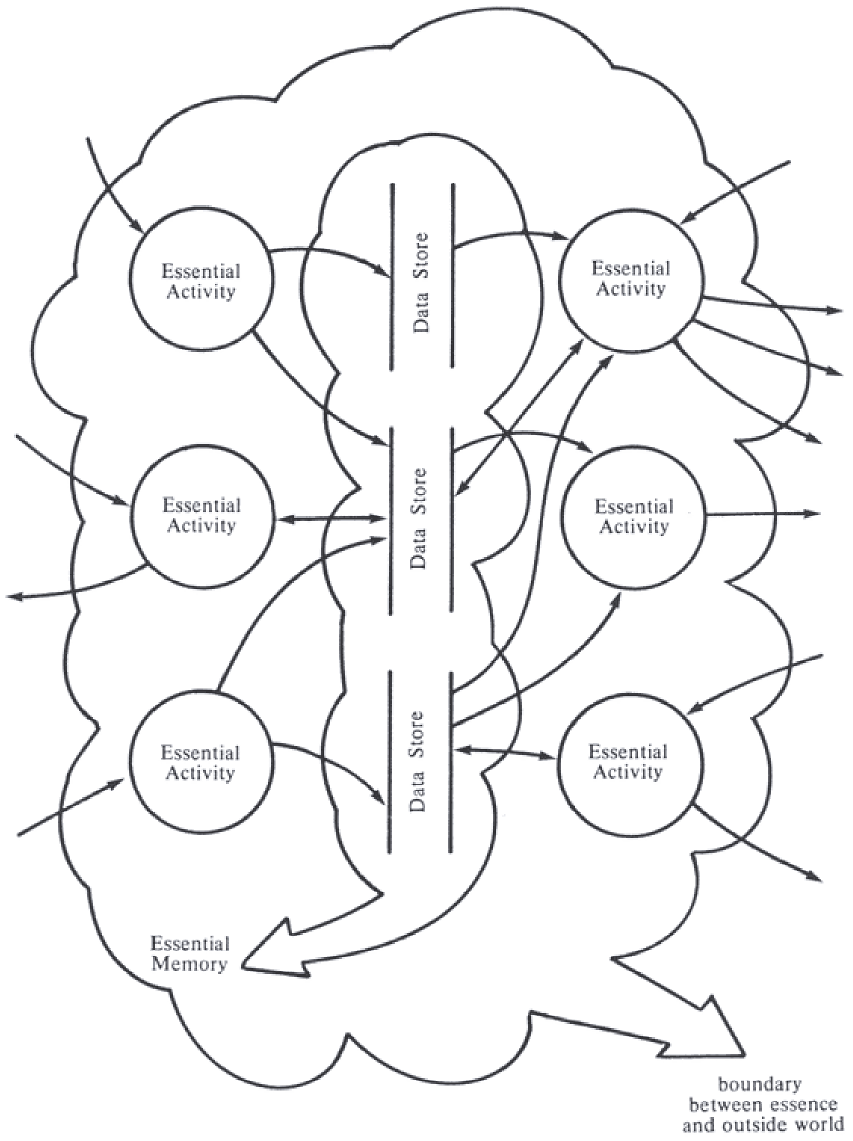
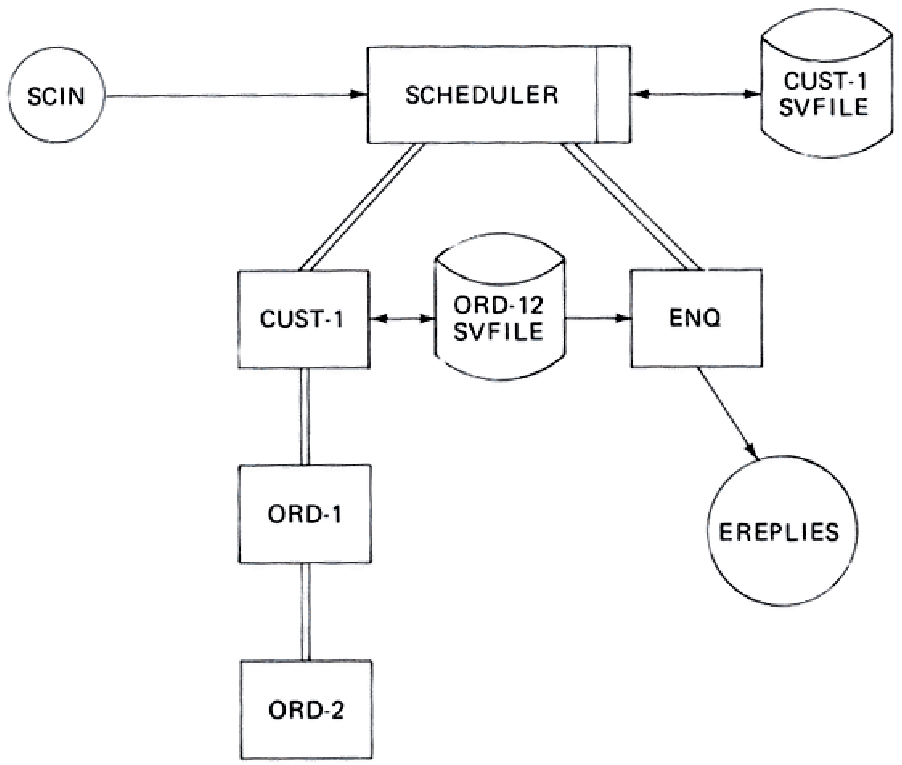


Figure 7.5. Characteristic shape of an event-partitioned DFD.

Basically, ESA views a computer system as a single, huge OO-style object. The system's essential activities are its methods, and its essential memory is its internal data. And just as an object is an example of the Headless Handlers pattern, with the methods playing the role of handlers, so here an entire computer system is an example of the Headless Handlers pattern, with the essential activities playing the role of handlers.

The most fully worked-out conceptualization of a system on the Handlers pattern was JSD (Jackson System Development), described in Michael Jackson's book *System Development* (1983). JSD was a brilliant method, and arguably the first true object-oriented analysis and design method.

The design of a JSD system is shown in an SID (system implementation diagram). Here is a typical SID, from p. 293 of *System Development*.



At the top of this diagram, we see the dispatcher — in JSD it is called the *scheduler*. A stream of events arrives from outside the system; it is labeled SCIN ("scheduler input"). Events are dispatched to CUST-1 or ENQ, the event handlers. The system's internal data is stored in database tables called "state vector" files (labeled SVFILE). EREPLIES is a stream of replies produced in response to operator enquiry events.

CUST-1 and ENQ are not functions; they are full-fledged OO-style objects. This means that JSD uses the Handlers pattern on two levels. On the upper level, the entire system is conceived on the full Handlers pattern, with the scheduler as the dispatcher and objects as event handlers. On the secondary level are OO-style objects, whose methods function as event handlers for events arriving from the scheduler or from other objects.

ESA and JSD mark a huge shift in thinking from the earlier structured methods. The cause of the shift was the rapid advances that were being made in database technology. In the early days of structured analysis, database management systems (DBMSs) were essentially non-existent. But by the time ESA and JSD appeared, computerized data processing was quickly moving away from batch systems processing sequential files to online systems processing databases. At first there were linked-list DBMSs (e.g. IBM's IMS, Cullinane's Cullinet, and Cincom's Total). These were quickly followed by inverted-list DBMSs (Adabas, Model204), then by relational DBMSs (DB2, Ingres, Oracle). The advance of DBMS technology was accompanied by the development of database design methodologies (ERA, the Entity-Relationship Approach; IE, Information Engineering; NIAM/ORM; IDEF1X).

As database technology improved and became more widely used, developers increasingly came to see the database – not the software that accessed it – as the heart of a computer system. The new model of a computer system – as a set of event handlers surrounding and providing an interface to the database that lay at the core of the system – was, therefore, very much a product of its time.

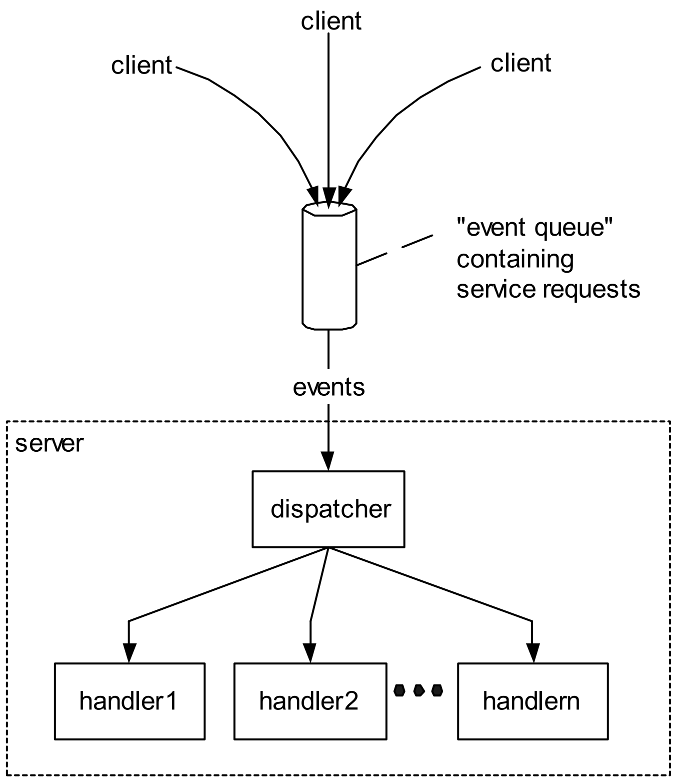
Client-Server Architecture

A familiar example of the Handlers pattern occurs in *client-server* architectures. A *server* is a piece of hardware or software that provides a *service* to *clients* that use the service. The server's job is to wait for *service requests* from clients, to respond to service requests by providing the requested service, and then to wait for more requests. Examples of servers include: print servers, file servers, windows servers, database servers, application servers, and web servers. If you've ever surfed the Web, you've interacted with a server – every time you surfed to a new web address, your web browser sent a service request to a web server, which responded by serving up the web page that you requested.

Wesley Chun provides a short, clear explanation of the basics of client-server architecture in *Core Python Programming,* Chapter 16. (I've slightly modified the text to improve consistency of the terminology.) Imagine, says Chun,

> a blank teller who neither eats, sleeps, nor rests, serving one customer after another in a line that never seems to end. The line may be long or it may be empty on occasion, but at any given moment, a customer may show up. Of course, such a teller was fantasy years ago, but automated teller machines (ATMs) seem to come close to the model now.
>
> The teller is, of course, the server that runs in an infinite loop. Each customer has a need *[a service request]* which requires servicing. Customers arrive and are serviced by the teller in a first-come-first-served manner. Once a transaction has been completed, the customer goes away while the server either serves the next customer or waits until one comes along.

Here is the situation, described in terms of the Handlers pattern.

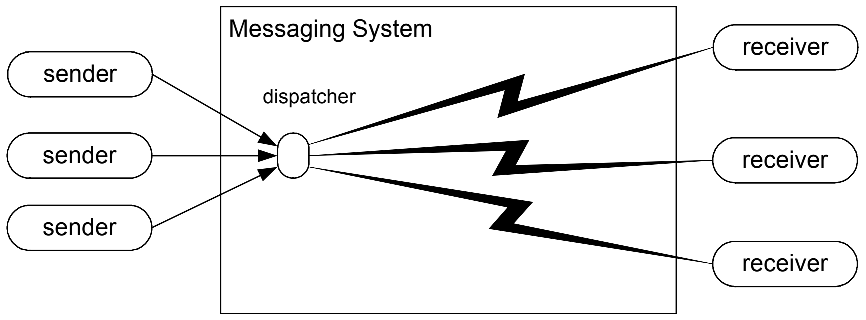


Each bank customer represents a service request (event, transaction) sent by a client.
The customers queue up and wait for service. The tireless teller is a good analogy for a server which, because of its collection of event handlers, can handle different kinds of service requests. And the bank teller's "infinite loop" is of course the dispatcher's event loop.

Messaging Systems

Messaging systems represent an extreme version of the Handlers pattern. The purpose of a messaging system is to get events (*messages*) from event generators (*senders)* to handlers (*receivers*) in situations where the senders and receivers are in different physical locations or running on different platforms.

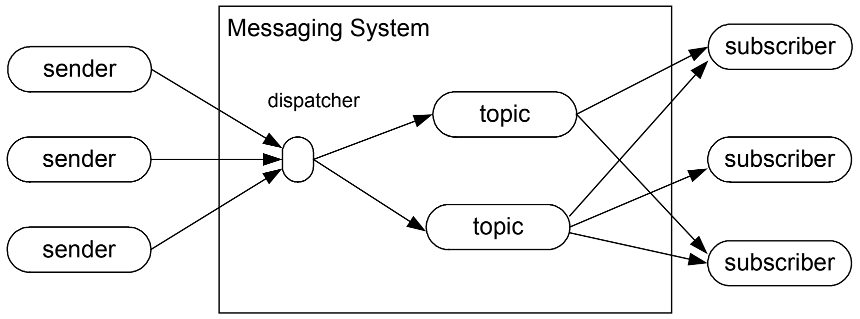
In messaging systems, messages are typically *addressed* to specific receivers, so the dispatching function (which determines which receiver should receive the message) is trivially simple. A familiar example of a messaging system is the post office. A sender gives a message (a letter or a parcel) to the post-office (the messaging system). The post office reads the receiver's address on the message and transports the message to the receiver.



E-mail messaging systems perform essentially the same function as the post office; the only difference is that the messages are encoded electronically rather than physically.

Perhaps the most sophisticated kinds of messaging systems are enterprise messaging systems, which use *message-oriented middleware* or MOM. In MOM systems, the senders and receivers are computer applications, rather than human beings. MOM systems enable computerized applications that are physically separated, or running on different hardware/software platforms, to communicate with each other. For example, a big company may have offices and servers that are geographically dispersed. MOM software could allow an order placed with the company's order-entry system in LA to be electronically sent to an order-fulfillment application hosted on a server in Chicago, and to a management-reporting application hosted on a server in New York, all without human intervention.

In addition to this *point-to-point* model of communications, MOM products also support a *publish/subscribe* model. In a publish/subscribe model, receivers become *subscribers* by subscribing to *topics*, and senders may send messages to topics (rather than to individual subscribers). When a topic receives a message, the topic forwards the message to all receivers who have subscribed to the topic.



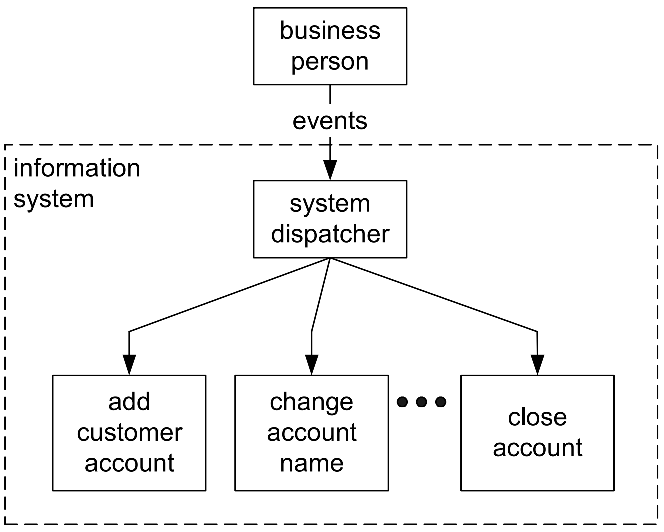
In MOM systems, telecommunications issues (and queuing issues, and implementation issues of the publish/subscribe model) dwarf the Handlers aspect of the system. Nevertheless, for some purposes it may be helpful to think of MOM systems as simply extreme, specialized examples of the Handlers pattern.

Frameworks

Object-Oriented Event-Driven Programming

Now that we've seen the Big Picture – how the Handlers pattern is manifested in many different aspects of modern computer systems – let's look at how the Handlers pattern works at the code level.

Consider a business that deals with customers. Naturally, the business owner wants an information system that can store, retrieve, and update information about his customer accounts. He wants a system that can handle a variety of events: requests to add a new customer account, to change the name on an account, to close an account, and so on. So the system must have event handlers for all of those types of events.



Before the advent of object-oriented programming, these event handlers would have been implemented as subroutines. The code inside the dispatcher's event loop might have looked like this:

```
get eventType from the input stream

if   eventType == EndOfEventStream :
     quit # break out of event loop

if   eventType == "A" : call addCustomerAccount()
elif eventType == "U" : call setCustomerName()
elif eventType == "C" : call closeCustomerAccount()

... and so on ...
```

And the subroutines might have looked something like this.

```
subroutine addCustomerAccount():
    get customerName                    # from data-entry screen
    get next available accountNumber    # generated by the system
    accountStatus = OPEN

    # insert a new row into the account table
    SQL: insert into account
         values (accountNumber, customerName, accountStatus)

subroutine setCustomerName():
    get accountNumber       # from data-entry screen
    get customerName        # from data-entry screen

    # update row in account table
    SQL: update account
         set   customer_name = customerName
         where account_num   = accountNumber

subroutine closeCustomerAccount():
    get accountNumber       # from data-entry screen

    # update row in account table
    SQL: update account
         set   status      = CLOSED
         where account_num = accountNumber
```

Nowadays, using object-oriented technology, event handlers would be implemented as methods of objects. The code inside the dispatcher's event loop might look like this:

```
get eventType from the input stream

if   eventType == "end of event stream":
     quit # break out of event loop

if   eventType == "A" :
     get customerName                     # from data-entry screen
     get next available accountNumber    # from the system

     # create an account object
     account = new Account(accountNumber, customerName)
     account.persist()

elif eventType == "U" :
     get accountNumber       # from data-entry screen
     get customerName        # from data-entry screen

     # create an account object & load it from the database
     account = new Account(accountNumber)
     account.load()

     # update the customer name and persist the account
     account.setCustomerName(customerName)
     account.persist()

elif eventType == "C" :
     get accountNumber       # from data-entry screen

     # create an account object & load it from the database
     account = new Account(accountNumber)
     account.load()

     # close the account and persist it to the database
     account.close()
     account.persist()

... and so on ...
```

And the *Account* class, with its methods that function as event-handlers, would look something like this.

```
class Account:

    # the class's constructor method
    def initialize(argAccountNumber, argCustomerName):
        self.accountNumber = argAccountNumber
        self.customerName  = argCustomerName
        self.status = OPEN

    def setCustomerName(argCustomerName):
        self.customerName = argCustomerName

    def close():
        self.status = CLOSED

    def persist():
        ... code to persist an account object

    def load():
        ... code to retrieve an account object from persistent storage
```

Using OO technology this way isn't very exciting. Basically, we've just substituted objects for database records; otherwise, the processing is pretty much unchanged.

But it gets more interesting...

Frameworks

With object-oriented technology it is relatively easy to develop generalized, reusable classes. This is one of the advantages of OO technology.

Suppose, for example, that there is a commercial, general-purpose business-support product called GenericBusiness. GenericBusiness is a software framework that knows how to perform a variety of common business functions – opening customer accounts, closing customer accounts, and so on. Naturally, because all businesses are different, GenericBusiness allows each business to customize the framework to meet that business's particular needs.

Suppose further that Bob is a small businessman and he buys a copy of GenericBusiness. Before he can use GenericBusiness, Bob needs to customize it for his particular needs. We can imagine a lot of things that Bob might want to customize: his business name, the names of the products he sells, the kinds of credit cards that he accepts, and so on. But for the sake of discussion, let's look at Bob's most pressing requirement – he wants to customize GenericBusiness to use MySQL[1] to store information about accounts.

GenericBusiness is ready. It knows that it can't predict which DBMS a business will want to use. This means that GenericBusiness – even though it knows how to open and close customer accounts – *doesn't* know how to persist account data to a database. And of course, it couldn't know. Without knowing what DBMS a business wants to use – Oracle, Sybase, DB2, MySQL, Postgres – GenericBusiness can't know what code to put in the *persist()* method of the Account class.

That means that Bob must write the code for the *persist()* method himself.

And *that* means that GenericBusiness has a problem – how can it insure that a customer like Bob will write the code for the *persist()* method?

The solution is for GenericBusiness not to supply a fully-functional *Account* class. Instead, it supplies a class that is, if you like, half-baked – it is a class that implements only *some* of the methods of a fully functional *Account* class. This partly-implemented class – let's call it the *GenericAccount* class – provides full implementations for some methods, and leaves *plug-points* where businessmen like Bob must add their business-specific code.

A plug-point is a place in the code where a software framework expects event-handlers to be "plugged in". The event-handlers themselves are called *plug-ins* (or, if you prefer, *plugins*).

The technical name for a half-baked class containing plug-points is an *abstract* class. Different programming languages provide different ways to define plug-points. Java, for example, provides the keyword "abstract" and plug-points are called *abstract methods*.

An abstract method isn't really a method. Rather, it is a place-holder for a method; a spot where a real (*concrete*) method can be plugged in. A Java class that contains abstract methods is called an *abstract class*. An abstract class can't be instantiated. The only way to use an abstract class is (a) to create a subclass that extends it, and (b) to have the subclass define real (*concrete*) methods

[1] MySQL is a relational DBMS – database management system.

that implement each of the abstract methods. Java itself enforces this requirement; the Java compiler won't compile a program that attempts to instantiate an abstract class.

This means that the way for Bob to use GenericBusiness's *GenericAccount* class is for him to create a concrete class that extends it, and implements the abstract methods. If the *GenericAccount* class looks like this...

```
# an abstract class
class GenericAccount:
    ...
    # an abstract method named "persist"
    def ABSTRACT_METHOD persist():
    # no code goes in an abstract method
```

Then Bob's *Account* class will look like this....

```
class Account(extends GenericAccount):
    ...
    # a concrete method named "persist"
    def persist():
        ... code that Bob writes, to persist an account object
        ...  to the DBMS of Bob's choice
```

Python, a dynamic language, supports plug-points and abstract classes in a different way than Java does. In Python, the simplest[2] way to implement a plug-point is to define a method that does nothing but raise an exception. If the method isn't over-ridden, and a program tries to invoke it, a run-time exception is triggered. Here's an example of the Python code for an abstract method.

```
def setCustomerName(self): # an abstract method, a plug-point
    raise Exception("Unimplemented abstract method: persist")
```

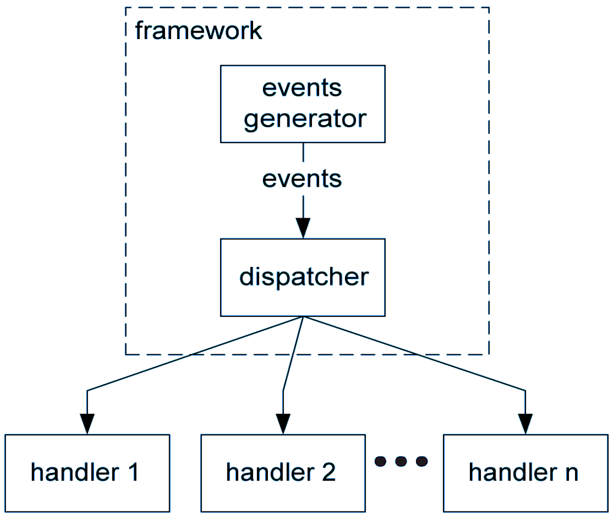
The general term for a piece of software that works this way – that defines plug-points and requires plug-ins – is *framework*. Here are some of the definitions you will get if you Google for the term "framework". Each of the definitions captures part of the concept of a framework. A framework is:

- a skeletal structure for supporting or enclosing something else
- a broad overview, outline or skeleton, within which details can be added
- an extensible software environment that can be tailored to the needs of a specific domain.
- a collection of classes that provides a general solution to some application problem. A framework is usually refined to address the specific problem through specialization, or through additional types or classes.

[2] Other possible ways to implement abstract methods in Python can be found in Appendix A.

- a component that allows its functionality to be extended by writing plug-in modules ("framework extensions"). The extension developer writes classes that derive from interfaces defined by the framework.
- a body of software designed for high reuse, with specific plugpoints for the functionality required for a particular system. Once the plugpoints are supplied, the system will exhibit behavior that is centered around the plugpoints.

The pattern underlying the concept of a framework is the Handlers pattern. The "framework extensions" or "plug-ins" are event handlers.[3]



SAX – an example of a framework

Frameworks come in all shapes and sizes, from very large to mini-frameworks. To see how a real framework is used, let's look at one of the smaller ones: SAX[4].

XML is surging in popularity. One consequence is that many programmers are encountering event-driven programming for the first time in the form of SAX – the Simple API for XML. SAX is an event-driven XML parser. Its job is to break (parse) XML into digestible pieces. For example, a SAX parser would parse the string:

```
<superhero>Batman</superhero>
```

into three pieces.

[3] The term "plug-in" or "plugin" is often used to denote downloadable components, built to perform specific tasks that can be plugged into a general-purpose tool. For example, if your Web browser doesn't know how to open files of a particular kind, you can download and install a "browser plugin" for that kind of file. The Eclipse IDE is a general-purpose "pluggable" framework into which it is possible to plug extensions for different programming languages. And so on.

[4] Actually, SAX is not a framework. It is an API that may be implemented in a framework. But to keep things simple, I'll talk as if it is a framework.

```
<superhero>Batman</superhero>
-----------......------------
     |        |       |
     |        |       |
     |        |       endElement
     |        |
     |        characters
     |
     startElement
```

To use a SAX parser, you feed it a chunk of XML. It parses the XML text into different kinds of pieces and then invokes appropriate predefined plug-points (event-handlers) to handle the pieces.

SAX specifies predefined plug-points for a variety of XML features such as opening and closing tags (startElement, endElement), for the text between tags, comments, processing instructions, and so on.

The SAX framework provides a *Parser* class and an abstract *ContentHandler* class. To use it, you first subclass the ContentHandler class and write concrete methods (event-handlers) to over-ride the abstract methods. Here is a simple example in Python. All it does is print (i.e. write to the console) the XML tag names and the data between the tags. (A fuller example of a Python SAX program can be found in Appendix B.)

```
# example Python code

class CustomizedHandler(ContentHandler):

    def startElement(self, argTag, argAttributess):
        # write out the start tag, without any attributes
        print "<" + argTag + ">"

    def endElement(self, argTag):
        print "<" + argTag + ">"

    def characters(self, argString):
        print argString
```

Once you've extended the ContentHandler class and specified the event handlers, you:

- use the SAX *make_parser* factory function to create a parser object
- instantiate the *CustomHandler* class to create a *myContentHandler* object
- tell the parser object to use the *myContentHandler* object to handle the XML content
- feed the XML text to the parser and let the event handlers to all the work.

Here is how it might be done in Python.[5]

```
myParser = xml.sax.make_parser()       # create parser object
myContentHandler = CustomizedHandler() # create content handler object

# tell the parser object which ContentHandler object
# to use to handle the XML content
myParser.setContentHandler(myContentHandler)

myInfile = open(myInfileName, "r") # open the input file
myParser.parse(myInfile)           # send it to the parser to be parsed
myInfile.close()                   # close the input file
```

Why programming with a framework is hard

Note that the last step in the process consists only of feeding the XML text to the parser – nothing more. For a programmer with a background in procedural programming, this is what makes event-driven programming confusing. In a procedural program, the main flow of control stays within the main application routine. Subordinate routines or modules are merely utilities or helpers invoked to perform lower-level tasks. The flow of control in the main routine is often long and complex, and its complexity gives the application a specific logical structure. The program has a shape, and that shape is visible to the programmer.

But when a procedural programmer begins programming with a framework, he loses all of that. There is no discernable flow of control – the main routine doesn't do anything except start the framework's event-loop. And once the event-loop is started, it is the code hidden inside the framework that drives the action. What's left of the program seems to be little more than a collection of helper modules (event-handlers). In short, the program structure seems to be turned inside out.[6]

So procedural programmers often find that, on first encounter, event-driven and framework-driven programming makes no sense at all! Experience and familiarity will gradually lessen that feeling, but there is no doubt about it – moving from procedural programming to event-driven programming requires a very real mental paradigm shift. This is the paradigm shift that Robin Dunn and Dafydd Rees were describing in the quotations at the beginning of this paper.

[5] Note that the specific details of how this is done will vary with the implementation language and the implementation of SAX. Sun's web site has a nice tutorial on SAX programming with Java at http://java.sun.com/webservices/docs/1.3/tutorial/doc/JAXPSAX.html

[6] This is the difference between a *framework* and a *library* (i.e. a collection of utilities). When using a library of utilities, a programmer shapes the control flow of the program, and calls utilities when he needs them. With a framework, the framework is in charge and calls the programmer-written event-handler modules when it needs them. It's a question of who's in charge.

GUI programming

Why GUI programming is hard

Now that we've seen how frameworks work, let's look at one of the most common uses for frameworks and event-driven programming: GUIs (graphical user interfaces).

GUI programming is hard; everybody admits it.

First of all, a lot of work is required simply to specify the appearance of the GUI. Every widget – every button, label, menu, enter-box, list-box, etc. – must be told what it should look like (shape, size, foreground color, background color, border style, font, etc.), where it should position itself, how it should behave if the GUI is resized, and how it fits into the hierarchical structure of the GUI as a whole. Just specifying the appearance of the GUI is a lot of work. (This is why there are IDEs and screen painters for many GUI frameworks. Their job is to ease the burden of this part of GUI programming.)

Second (and more relevant to the subject of this paper), GUI programming is hard because there are many, *many* kinds of events that must be handled in a GUI. Almost every widget on the GUI – every button, checkbox, radio button, data-entry field, list box (including every item on the list), textbox (including horizontal and vertical sliders), menu bar, menu bar icon, drop-down menu, drop-down menu item, and so on – almost every one is an event generator capable of generating multiple kinds of events. As if that weren't enough, hardware input devices are also event generators. The mouse can generate left button clicks, right button clicks, left button double clicks, right button double clicks, button-down events (for initiating drag-and-drop operations), mouse motion events, button-up events (for concluding drag-and-drop operations), and others. Every letter key on the keyboard, every number key, punctuation key, and function key – both alone and in combination with the SHIFT, ALT, and CONTROL keys – can generate events. There are *a lot* of kinds of events that can be fed into the dispatcher's event loop, and a GUI programmer has to write event handlers for every significant event that a GUI-user might generate.

Third, there is the fact that many GUI toolkits are supplied as frameworks. The purpose of GUI frameworks, like the SAX framework, is to ease the burden on the GUI programmer. GUI frameworks, for instance, supply the event-loop and the event queue, relieving the GUI programmer of that job. But, as we saw with SAX, using a framework means that large chunks of the control flow of the program are hidden inside the canned machinery of the framework and are invisible to the GUI programmer. This means that a GUI programmer must master the paradigm shift involved in moving to event-driven programming.

It's a lot to deal with. And, as if that weren't enough, there is the *Observer* pattern to master…

The *Observer* Pattern

The Observer pattern is widely used in doing event-driven programming with GUI frameworks. So now I am going to make a detour to explain the Observer pattern. Once that is done, I'll return to the topic of GUI programming, and show how the Observer pattern is used in GUI programming.

The Observer pattern was first named and described in the famous "Gang of Four" book, *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (Addison-Wesley, 1995). The basic idea underlying the Observer pattern is the principle that Dafydd Rees referred to in the quotation in the introduction. The principle is *the Hollywood Principle:* "Don't call us; we'll call you." The name, of course, comes from the situation in which actors audition for parts in plays or movies. The director, who is casting the movie, doesn't want to be plagued by calls from all of the actors (most of whom, of course, didn't get the part), so they are all told, "Don't call us; we'll call you."

The longer version of the Hollywood Principle is:

> "Don't call us. Give us your telephone number, and we will call you if
> and when we have a job for you."

And that, in essence, is the Observer pattern.

In the Observer pattern, there is one *subject* entity and multiple *observer* entities. (In the audition scenario, the director is the subject and the actors are the observers.) The observers want the subject to *notify* them if *events* happen that might be of interest to them, so they *register* (leave instructions for how they can be reached) with the subject. The subject then notifies them when interesting events (such as casting calls) happen. To support this process, the subject keeps a list of the names and addresses of all of the observers that have registered with him. When an interesting event occurs, he goes through his list of observers, and notifies the observers who registered an interest in that kind of event.

The Observer pattern is also known as the *Publish/Subscribe* pattern. Think of the process of subscribing to a newspaper as an example. In the Publish/Subscribe pattern, the subject is the *publisher* of some information or publication. The observers are the *subscribers* to the publication. The process of registration is called *subscription*, and the process of notification is called *publication*. "Publish/Subscribe" is an appropriate name to use in situations where notification/publication occurs repeatedly over a long period of time, the same notifications are sent to multiple subscribers, and subscribers can *unsubscribe*.

The Observer pattern is a special case of the Handlers pattern, and for that reason I like to think of it as the *Registered Handlers* pattern.

On the next page is the Python code for a simple implementation of the Observer pattern. In this example, in keeping with the Hollywood theme, the observers are actors. The subject is a talent agency called HotShots. Actors register with the talent agency, and whenever there is a casting call (an audition) for the kind of role for which an actor has registered an interest, the agency notifies the actor. Since it may do this repeatedly, and for many actors, this example has something of a Publish/Subscribe flavor.

(If you're a Java programmer, please don't be distracted by the fact that we barely define the CastingCall and Observer classes. This is possible in a dynamic language like Python, and I've done it here to keep the code short. In Java, of course, you'd have to declare these classes and their instance variables in detail.)

[Look at the code now]

As you can see from this code, the talent agency (the subject) is itself an event handler. Specifically, its `notify` method is an event handler for a `CastingCall` event.

The talent agency handles `CastingCall` events by sending them to its own `notifyActors` method. The `notifyActors` method then looks through the agency's list of subscribers/observers/actors and sends out notifications to the appropriate subscribers.

In a real application, the notification process would consist of contacting each actor (event handler), who would respond by going to the casting call. In this simple example, however, the notification process consists only of printing a message saying that the actor has been notified.

If you ran this program, the output would be:

```
Larry got call for: male lead
Moe got call for: male lead
```

```
#  A short Python program showing the Observer pattern

class TalentAgency:      # define a TalentAgency class: the subject

    def __init__(self):                # The talent agency constructor.
       self.ourActors = []             # Initially our list of actor/subscribers is empty.


    def registerActor(self, argActor, argDesiredTypeOfRole ):
        observer = Observer()          # The agency creates a new observer object
        observer.actor       = argActor
        observer.desiredRole = argDesiredTypeOfRole

                                # add the observer to the agency's list of actor/subscribers
        self.ourActors.append( observer )


    def notify(self, castingCall):        # HotShots has been notified of a casting call.
        self.notifyActors(castingCall)    # Notify our clients!


    def notifyActors(self, castingCall):

        for observer in self.ourActors:      # Look at each actor on our list of clients.
                                             # If the actor wants roles of this type...
            if observer.desiredRole == castingCall.role:
                                             # ...notify the actor of the casting call
                print observer.actor, " got call for: ", castingCall.role


class CastingCall: pass        # define a CastingCall class
class Observer   : pass        # define an Observer class


#   Now let's run a little demo...

HotShots = TalentAgency()             # We create a new talent agency called HotShots


HotShots.registerActor( "Larry", "male lead" )       # actors register with HotShots,
HotShots.registerActor( "Moe"   , "male lead" )      # indicating the kind of role that
HotShots.registerActor( "Curly", "comic sidekick" ) # they are interested in.


castingCall        = CastingCall()          # There is a casting call --
castingCall.source = "Universal Studios"    # Universal Studios is looking
castingCall.role   = "male lead"            # for a male lead


HotShots.notify(castingCall)                # Hotshots gets notified of the casting call
```

Event Objects

There is one feature of this program that you should pay special attention to. It is the use of CastingCall objects with attributes (instance variables) of *source* and *role*. Here, the CastingCall objects are *event objects* – objects for holding events.

Event objects are wonderful tools for doing event-driven programming. In pre-object-oriented programming languages, events or transactions were extremely limited. In many cases, if you sent an event to an event handler, all you were sending was a single string containing a transaction code. But object-oriented technology changed all that by allowing us to create and pass *event objects*. Event objects are essentially packets into which we can stuff as much information about an event as we might wish.

An event object will of course carry the name of the kind of event that triggered it. But, depending on the application, it might carry much more. In the case of the HotShots talent agency, the event objects contain information about the casting call's source and role.

In other applications, we might want to put quite a lot of information into our event objects. Consider for instance the famous Model-View-Controller (MVC) pattern.[7] The Observer pattern is really the heart of MVC. In MVC, the *Model* is an object that manages the data and behavior of some application domain: it is the subject in the Observer pattern. *Views* register with the Model as observers. Whenever the *Controller* makes a change to the Model, the Model notifies its registered observers (the Views) that it (the Model) has changed.

The simplest version of MVC is called the *pull* version. In this version the event object (the change notification that the Model sends to the Views) contains hardly any information at all. The Model doesn't describe the change, it only notifies the Views that some kind of change has taken place. When the Views receive such a notification, they must then *pull* information from the Model. That is, they must query the Model for information about its current state, and refresh themselves from that information.

The more elaborate version of MVC is called the *push* version. In this version, the Model *pushes* out change information to the Views. The event object sent to the Views contains a great mass of information – a complete and detailed description of the change that has been made to the Model. When the Views receive this information, they have all the information they need in order to modify themselves, and they do.

The basic difference between the *push* and *pull* versions of MVC is simply the amount of information that is stuffed into the event-object packet.

[7] See *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)* by Steve Burbeck (original version, 1987) at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

The *Registered Handlers* pattern in GUI applications

Now that you are familiar with the basics of the Observer/RegisteredHandlers pattern, let's look at how the pattern is used in GUI programming.

Here's the code for another program. Structurally, the code is very similar to the code for the Hotshots talent agency, but the methods have been renamed to use the terminology of the Handlers pattern, and the events are GUI events.

In this example, the "subject" is the dispatcher that works inside the event-loop of a GUI.

To keep the code short, the program contains only one observer – the `demoHandler` function is an event-handler for a double click of the left button of the mouse.

To demonstrate the action of this handler, the program generates a simulated `LeftMouseDoubleClick` event.

[Look at the code now]

If you ran this program, the output would be:

```
Handling LeftMouseDoubleClick from mouse
```

Note that in statements like this:

```
demoDispatcher.registerObserver( demoHandler, MOUSE_LEFT_DOUBLE )
```

and like this:

```
observer.eventHandler = argEventHandler
```

the program is passing around a reference to the **`demoHandler`** function object. That is, it is treating a function as a first-class object. This isn't possible in all programming languages, which means that the implementation of the Observer/RegisteredHandlers pattern can vary considerably between programming languages.

```
# A short Python program showing the Registered Handlers pattern

class Event   : pass
class Observer: pass

# Create an event handler (i.e. an observer).  In this case, the event handler is a function.
# It simply prints information about the event that it is handling.
def demoHandler(argEvent):
    print "Handling", argEvent.type, "from", argEvent.source


class Dispatcher:    # Define the subject

    def __init__( self ):                   # __init__ = a Python constructor
        self.myObservers = []               # initialize a list of observers.


    def registerObserver( self, argEventHandler, argEventType ):
        # Register an observer.  The argEventType argument indicates the event-type
        # that the observer/event-handler will handle.

        observer = Observer()                      # Create an "observer" object
        observer.eventHandler = argEventHandler    # and pack the event-handler
        observer.eventType    = argEventType       # and event-type into it.

        self.myObservers.append( observer )        # add observer to the "notify list"


    def eventArrival(self, argEvent):
        self.notifyObservers(argEvent)      # dispatch event to registered observers


    def notifyObservers( self, argEvent):
        for observer in self.myObservers:
            # if the handler wants to handle events of this type...
            if observer.eventType == argEvent.type:
                # ...send the event to the handler
                observer.eventHandler(argEvent)


# Run a demo.  We simulate a GUI user doing a mouse-click.

demoDispatcher = Dispatcher()    # create a dispatcher object

MOUSE_LEFT_DOUBLE = "LeftMouseDoubleClick"   # define an event type

# Register the event handler with the dispatcher.  The second argument
# specifies the type of event to be handled by the handler.
demoDispatcher.registerObserver( demoHandler, MOUSE_LEFT_DOUBLE )


demoEvent        = Event()               # Create an event object for demo purposes
demoEvent.type   = MOUSE_LEFT_DOUBLE     # The event is a left double-click
demoEvent.source = "mouse"               # of the mouse

demoDispatcher.eventArrival(demoEvent)   # Send the mouse event to the dispatcher
```

Registering Event-Handlers in Python – "binding"

These are the basic ideas behind the *Registered Handlers* pattern. Now let's see what the Registered Handlers pattern looks like in GUI applications in Python and Java.

Open-source dynamic languages such as Python, Perl, and Ruby usually support GUI programming by providing interfaces to a variety of open-source GUI frameworks. Python, for example, provides several such interfaces. The most popular are tkinter (which provides an interface to Tcl/Tk) and wxPython (which provides an interface to wxWidgets).

The basic concepts used in these interfaces are all pretty similar. In the following discussion, I will use examples from Python and tkinter to illustrate one style of using the Registered Handlers pattern in GUI programming.

In Python and tkinter, all GUI events belong to a single class called *event*. Event-handlers are registered with GUI widgets (buttons, and so on) for handling particular types of events such as mouse clicks or key presses. In Python, the process of registering an event-handler is called *binding*.

Here's a simple example. Assume that our program already has an event-handler routine (a function or method) called OkButtonEventHandler. Its job is to handle events that occur on the "OK" button on the GUI.

Note that in Python there is nothing special or magic about the name "OkButtonEventHandler" – we could have named the routine "Floyd" or "Foobar" if we had wished. As we shall see, this is one way in which Python differs from Java.

The following code snippet creates the "subject" widget in the Observer pattern. The subject is a GUI widget – a button – that displays the text "OK". The OkButton object is an instance of the Tkinter.Button class.

```
OkButton = Tkinter.Button(parent, text="OK")
```

The "parent" argument on the call to the Button class's constructor links the button object to its owner GUI object (probably a frame or a window).

Tkinter widgets provide a method called *bind* for binding events to widgets. More precisely, the *bind* method provides a way to "bind" or associate three different things:

- a type of event (e.g. a click of the left mouse button, or a press of the ENTER key on the keyboard)
- a widget (e.g. a particular button widget on the GUI)
- an event-handler routine.

For example, we might bind (a) a single-click of the left mouse button on (b) the "CLOSE" button (widget) on a window to (c) a *closeProgram* function or method. The result would be that when a user mouse-clicks on the "CLOSE" button, *close Program* is invoked and closes the window.

Here's a code snippet that binds a combination of the OkButtonEventHandler routine and a keyboard event ("<Return>") to the OkButton widget:

```
OkButton.bind("<Return>"  , OkButtonEventHandler)
```

This statement *registers* the OkButtonEventHandler function with the OkButton widget as the *observer* for keyboard "<Return>" events that occur when the OK button object has keyboard focus.

Here's another code snippet (that might occur in the same program, right after the previous code snippet). It binds the OkButtonEventHandler routine and a left mouse-click event to the OkButton widget:

```
OkButton.bind("<Button-1>", OkButtonEventHandler)
```

If either of these events[8] occurs, the event will send an event object as an argument to the OkButtonEventHandler function. The OkButtonEventHandler function can (if it wishes) query the event object and determine whether the triggering event was a key press or a mouse click.

[8] a keyboard "<Return>" event when the OK button object has keyboard focus, or a left mouse-click event on the "OK" button

Registering Event-Handlers in Java – "listeners"

Java also supports techniques for registering GUI event handlers, but its approach is quite different from Python's.

Java provides a good selection of capabilities for GUI programming in AWT and Swing. These capabilities allow a programmer both to create the visual layout of the GUI and to listen for events from the GUI.

For the event-handling aspects of the GUI, the java.awt.event package provides a number of different types of event-object:

ActionEvent	InvocationEvent
AdjustmentEvent	ItemEvent
ComponentEvent	KeyEvent
ContainerEvent	MouseEvent
FocusEvent	MouseWheelEvent
InputEvent	PaintEvent
InputMethodEvent	TextEvent

Each of these event types contains variables and methods appropriate for events of that type. For example, *MouseEvent* objects contain variables and methods appropriate for mouse events, and *KeyEvent* objects contain variables and methods appropriate for keyboard events. For example:

- MouseEvent.*getButton()* reports which mouse button caused the event
- MouseEvent.*getClickCount()* reports the number of mouse clicks that triggered the event
- MouseEvent.*getPoint()* tells the x,y coordinates of the position of the mouse cursor within the GUI component
- KeyEvent.*getKeyChar()* tells which key on the keyboard was pressed.

The java.awt.event package also provides a generic *EventListener* interface and a collection of specialized listeners that extend it. Examples of the specialized listener interfaces are:

ActionListener	MouseListener
ContainerListener	MouseMotionListener
FocusListener	MouseWheelListener
InputMethodListener	TextListener
ItemListener	WindowFocusListener
KeyListener	WindowListener

These specialized listener interfaces are built around the different event types. That is, a listener interface is a collection of methods (event handlers), all of which handle the same type of event object. The methods in the MouseListener interface, for instance, handle MouseEvents; the methods in the KeyListener interface handle KeyEvents, and so on.

Within the interfaces, the event handlers have descriptive, hard-coded names. For example, the MouseListener interface provides five event-handler methods:

- mouseClicked(MouseEvent e)
- mouseEntered(MouseEvent e)
- mouseExited(MouseEvent e)
- mousePressed(MouseEvent e)
- mouseReleased(MouseEvent e)

A GUI consists of multiple GUI components (widgets) such as panels, lists, buttons, etc. When a GUI program runs, the widgets are where the GUI events – mouse clicks, key presses, etc. – originate.

In terms of the Observer, pattern the widgets are the "subjects". Each widget, therefore, must provide some way for observers to register with it. In Java, this is accomplished by having each of the Java GUI classes (JPanel, JButton, JList, etc.) provide methods for registering observer objects. JPanel provides an *addMouseListener()* method for registering observers of mouse events; JButton provides *addActionListener()*; and so on.[9]

To put up a GUI, a Java program must do the following tasks:

- Create and position the visual components of the GUI, the widgets.
- Create one or more listener objects – objects that implement all of the event-handler methods in the appropriate listener interface.
- Register the listener objects with the appropriate subject widgets, using the widgets' *add[xxx]Listener()* method.

When a GUI event – for example, a mouseClicked event – occurs on a subject widget, the widget will call the *mouseClicked()* event-handler method in the registered listener object and pass the mouseClicked event object to it.

Here is some example code based on Sun's tutorial "How to Write a Mouse Listener".[10]

[9] These methods are obtained via inheritance from ancestor classes – Jpanel inherits *addMouseListener()* from java.awt.Component, JButton inherits *addActionListener()*from javax.swing.AbstractButton, and so on.

[10] At http://java.sun.com/docs/books/tutorial/uiswing/events/mouselistener.html.
Specifically, the code is adapted from the MouseEventDemo class at http://java.sun.com/docs/books/tutorial/uiswing/events/example-1dot4/MouseEventDemo.java.

```
// We define a class.  This class is a mouse listener because
// it implements the MouseListener interface.
// It also extends JPanel, so it is also a GUI widget object.

public class DemoGUI extends JPanel implements MouseListener {


    public DemoGUI() {      // the constructor for DemoGUI
        ... create inputArea and reportingArea widgets/objects ...
        ... add them to our GUI ...
        // Tell the inputArea to add this object to its list
        // of mouse listeners, so when a mouse event occurs
        // in the inputArea widget, the event will be reported.

        inputArea.addMouseListener(this);
    }

    public void mouseClicked(MouseEvent e) {

        // Over-ride the mouseClicked method.
        // This method will be called when a mouseClicked event occurs.

        this.report(
            "mouseClicked event ("  + e.getClickCount() + " clicks)"
            , e  // NOTE: second argument is the MouseEvent object, e
            )
    }

    void report(String eventDescription, MouseEvent e) {

        // Display a message reporting the mouse event.

        reportingArea.append( eventDescription
            + " detected on "
            + e.getComponent().getClass().getName()
            + ".\n");
    }

} // end class definition
```

In this code, the *inputArea* object is the event generator. The *mouseClicked()* method is the event handler. It is also an example of the Registered Handlers/Observer pattern. *inputArea* is the subject in the Observer pattern, and the line:

```
inputArea.addMouseListener(this);
```

registers "this" (that is, DemoGUI) with *inputArea* as an observer.

One interesting thing about this code is that it shows how multiple inheritance is done in Java. In this example, the DemoGUI class is both a GUI widget and a listener. It is a JPanel – it inherits from JPanel – so it is a GUI container object for the inputArea widget. In addition, it is a listener – it implements the methods (e.g. MouseClicked) in the MouseListener interface.

Note that in Java, the event-handler methods implement methods defined in a listener interface, so the names (e.g. "mouseClick") are determined by the interface and the programmer has no choice in the matter. This contrasts with Python, where (as we've seen) a programmer can name event-handler methods in whatever way he/she sees fit.

Callback programming

When you read the documentation for GUI frameworks, you will notice that the observer/event-handlers may be referred to as *callbacks* because subject widgets "call back" to them to handle events. So you will often see this type of programming referred to as *callback* programming.

GUI programming – summary

This wraps up our discussion of event-driven programming for GUI applications. Or should I say – our discussion of the Handlers pattern in the context of GUI programming.

Fundamentally, GUI programming isn't very different from the other examples of the Handlers pattern that we've seen. The one thing that makes GUI programming different from other forms of the Handlers pattern is that it almost always involves the Observer pattern. That is: GUI programs almost always involve a registration or binding process in which event-handlers (observers) are bound to (registered with) event generators (subjects).

This process of registration seems to me to be a very minor variation on the basic Handlers pattern – it is just one particular way of associating event handlers with event generators.

Maintaining State

Many event-driven applications are *stateless*. This means that when the application finishes processing an event, the application hasn't been changed by the event.

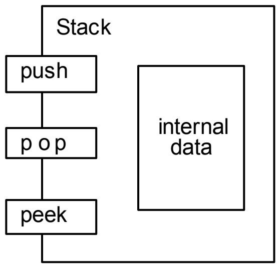
The opposite of a stateless application is a *stateful* application. Stateful applications are applications that *are* changed by the events that they process. Specifically, stateful applications remember or maintain information between events, and what they remember – their *state* information – can be changed by events.

A hammer is a stateless tool; if you drive a nail with it, the hammer is no different after you have driven the nail than it was before. A stapler is a stateful tool; if you staple some papers with it; the action of stapling changes its state – after the stapling action, the stapler contains one less staple than it did before.

In the most basic kind of Web browsing, a Web server receives a request to display a particular Web page, returns the requested page, and remembers nothing about what it has just done. This is a stateless application. A more sophisticated Web page might display a counter ("This page has been accessed 876,532 times since January 1, 2000"). To do this, the application behind the web page must remember the value of a counter and increment the counter every time the page is accessed. This application is stateful.

Rejecting invalid transactions

Earlier in our discussion we noted that object-oriented programming is a kind of event-driven programming. In most cases, object-oriented programming involves objects that are stateful. An object such as a stack maintains its state in its instance variables – the area labeled "internal data" on this diagram.



When we instantiate the Stack class to create a stack object, the stack starts out empty.

Now suppose that the follow sequence of events occurs:

- A *push(X)* event arrives. X is added to the stack, changing the stack's state.
- A *pop()* event arrives. X is removed from the stack (again changing the stack's state) and returned. The stack is now empty.
- Another *pop()* event arrives.

Now we have a problem. The stack cannot honor this request. It can't remove and return the topmost member of the stack because there is no topmost member of the stack; the stack is empty.

This example illustrates an important feature of stateful applications. Generally speaking, stateful applications define (at least implicitly) a list of acceptable *state + transaction type* pairs. For each pair on the list, the application – when it is in the specified state – can process a transaction of the specified type. But if an arriving transaction is <u>not</u> acceptable in the application's current state, the transaction must be rejected.

There are a lot of familiar examples of this sort of behavior:

- Generally speaking, an adult person can get married if he wants to... but he can't get married again if he's already married (to X number of spouses, where X is culturally determined).
- Generally speaking, you can deposit and withdraw money from your bank account... but you can't withdraw more money than you have in your account.
- Generally speaking, you can redeem your airline frequent-flyer miles for a free flight... but the airline won't allow you to do that during "blackout periods" of heavy travel, such as the day before Thanksgiving.
- Generally speaking, a database application can update records in the database... but it can't update a record that is locked by another application.

There are a variety of ways that a computerized application can respond to an invalid transaction. The simplest, of course, is simply to ignore it. But generally speaking, the best way to respond is to raise an exception and let the module that submitted the transaction deal with the exception.

Depending on the capabilities of the programming language being used, the application can raise a generic exception with a descriptive error message, or it can define specific kinds of exceptions for specific kinds of problems. A Stack, for instance, might define:

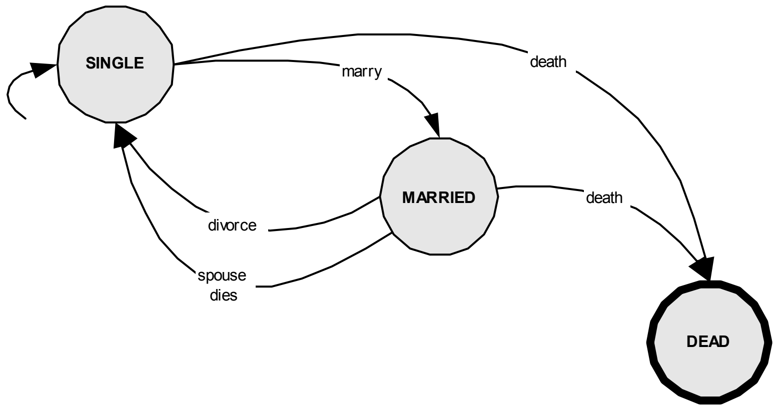
- a *StackEmptyException* that is raised when the stack is empty and it receives a *Pop()* request
- a *StackOverflowException* that is raised when the stack is full to capacity and it receives a *Push()* request.

State Machines

In many cases, it is useful to think of an object – a stateful computer application – as having a *lifecycle*. During its lifecycle, the stateful object moves from state to state in response to transactions (events).

An object that behaves this way is called a Finite State Machine (FSM) or a Directed Finite Automaton (DFA). The typical way to describe a finite state machine is with a State Transition Diagram (STD). On an STD, states are represented as circles and transitions are represented by arrows labeled with the name of the event that causes the transition.

Here is an example of an STD that represents the life and marriage history of a person (in a culture that permits a person to be married to only one spouse at a time).



The states in this STD are SINGLE, MARRIED, and DEAD. When a person begins his life, he is single (the *start state* is SINGLE). He may die without ever having been married, or he may marry. He may die while being married, or he may revert to his single status through divorce or the death of his spouse. DEAD is a *terminal state* – there are no event transitions out of that state.

Pseudo-code for such a Person class might look like this.

```
class Person:
    def __init__():
        self.status = "SINGLE"
        self.marriageCounter = 0

    def getMarried():
        if self.status == "SINGLE":
            self.status = "MARRIED"
            self.marriageCounter = self.marriageCounter + 1
        else:
            raise InvalidTransaction(self.status, "getMarried")

    def getDivorced():
        if self.status == "MARRIED":
            self.status = "SINGLE"
        else:
            raise InvalidTransaction(self.status, "getDivorced")

    def spouseDies():
        if self.status == "MARRIED":
            self.status = "SINGLE"
        else:
            raise InvalidTransaction(self.status, "spouseDies")

    def die():
        if self.status == "DEAD":
            raise InvalidTransaction(self.status, "die")
        else:
            self.status = "DEAD"
```

Note that earlier in our discussion of state, we used the expressions "state" and "state information" to refer to the whole body of information that a stateful object remembers – to the complete collection of all of an object's instance variables. But when talking of STDs and FSMs we used the word "state" to refer to specific, discrete states in a finite state machine.

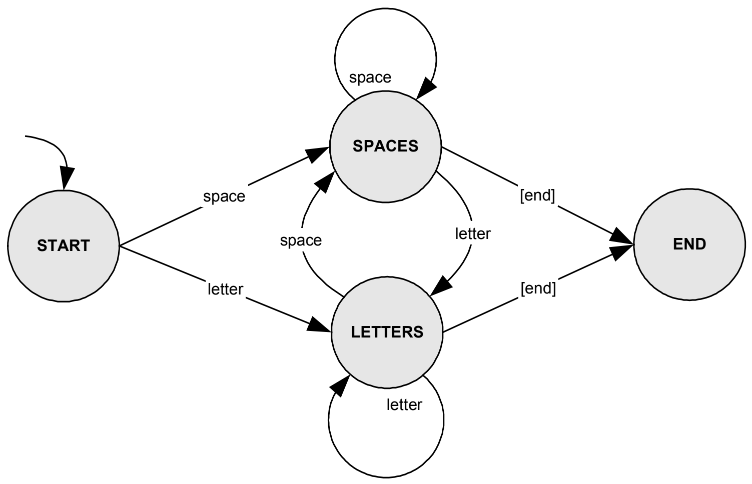
Unfortunately, there doesn't seem to be any standard terminology for distinguishing these two senses of the word "state". Michael Jackson uses the term "state vector" for "state" in the first sense – a "vector" is a list of variables that contain the state information of an object. Programmers often use the word "status" (as in "status_flag", "status_indicator") for "state" in the second sense.

In the rest of this paper, generally the context will indicate which sense of "state" I'm using. When there is a possibility of confusion, I will use the expressions "state vector" (or "state information") and "status" to keep things clear.

Coding a Finite State Machine (1)

There are many different techniques for implementing a finite state machine in code. In any given case, which technique you choose will depend on the nature of your application and the features of your programming language. So at this point, I'd like to show you one typical example of the implementation of an FSM program.

Here is the state transition diagram for a finite state machine whose purpose is to break an input stream of characters into groups of letters and groups of spaces.



On the next few pages is the source code for a program that implements this FSM. Before I show it to you, I'd like to point out a few things about it.

First of all, it is a parsing program. That is, it reads an input stream of characters, breaks it up into groups of characters (tokens), and labels the tokens by type. Parsing – which is done in the tokenizing (lexical analysis) phase of compilers, and in processing markup languages such as HTML and XML – is the classic application for illustrating the implementation of a finite state machine.

Second, it uses a classic design pattern for coding finite state machines. For many applications, a FSM program must do three things when processing a transaction:

- activities associated with leaving the current state
- change the current status to the new status
- activities associated with entering the new state

I've put comments in the code at the places where these activities should occur.

Finally, the program shows the features that we expect to see in the Handlers pattern. The input stream is the event queue. Reading a character from the input stream is the plucking of an event off of the event queue. An event loop examines each character/event/transaction to determine whether it is a letter or a space, and dispatches it to the appropriate handler. And of course there is code to break out of the event loop when an *EndOfEvents* (end of input stream) event is detected.

Here is the program. It is written in Python.

```
#--------------------------------------------------
#     some infrastructure
#--------------------------------------------------
# Python-specific code to get a character from the eventStream
def GetChar():
    for character in eventStream: yield character
    yield None # ... when there are no more characters
getchar = GetChar()

START   = "start..:"  # constants for state names
SPACES  = "spaces.:"
LETTERS = "letters:"
END     = "end....:"

def quote(argString): return '"' + argString + '"'


#-----------------------------------------------------
#   the event-handlers
#-----------------------------------------------------

def handleSpace(c):
    global state, outstring

    if state == START:
        # activities for exiting the current state
        # -- nothing to do when leaving START state

        # change the status to the new state
        state = SPACES

        # activities for entering the new state
        outstring = c

    elif state == SPACES:
        # activities for exiting the current state
        # -- do nothing: new state is same as old state

        # change the status to the new state
        # -- do nothing: new state is same as old state

        # activities for entering the new state
        outstring = outstring + c

    elif state == LETTERS:
        # activities for exiting the current state
        print state, quote(outstring)

        # change the status to the new state
        state = SPACES

        # activities for entering the new state
        outstring = c
```

```
def handleLetter(c):
    global state, outstring

    if state == START:
        # activities for exiting the current state
        # -- nothing to do when leaving START state

        # change the status to the new state
        state = LETTERS

        # activities for entering the new state
        outstring = c

    elif state == LETTERS:
        # activities for exiting the current state
        # -- do nothing: new state is same as old state

        # change the status to the new state
        # -- do nothing: new state is same as old state

        # activities for entering the new state
        outstring = outstring + c

    elif state == SPACES:
        # activities for exiting the current state
        print state, quote(outstring)

        # change the status to the new state
        state = LETTERS

        # activities for entering the new state
        outstring = c



def handleEndOfInput():
    global state, outstring

    if state == START:
        raise Exception("ERROR: Input stream was empty!")

    else:
        # activities for exiting the current state
        print state, quote(outstring)

        # change the status to the new state
        state = END

        # activities for entering the new state
        # -- nothing to do to startup END state
```

```
#-----------------------------------------------
# the driver routine, the event loop
#-----------------------------------------------

# Create an eventStream so we can demo the application
eventStream = "Suzy Smith loves       John Jones"

state = START # initialize the state-machine in the START state

while True: # do forever: this is the event loop

    c = getchar.next()       # get the character (event)

    if c == None:            # indicates end of the event stream
        handleEndOfInput()
        break                # break out of the event loop

    elif c == " ":
        handleSpace(c)

    else:                    # a "letter" is any non-space character
        handleLetter(c)
```

In this very simple example, the program takes the string

```
"Suzy Smith loves       John Jones"
```

... and produces the following output.

```
letters: "Suzy"
spaces : " "
letters: "Smith"
spaces : " "
letters: "loves"
spaces : "       "
letters: "John"
spaces : " "
letters: "Jones"
```

Coding a Finite State Machine (2)

As a practical matter, you will find that you use the concepts of:

- entering a state (status)
- maintaining state information (the state vector)
- leaving a state (status)
- rejecting transactions that are invalid in the current status

in almost every kind of event-driven application that you might choose to write.

- In GUI applications, widgets often need to remember their state. If a user clicks on a check-box, a GUI application will need to know the check-box's current state (checked, unchecked) in order to switch to the alternative state.

- Online applications need to remember all kinds of state information. If a user tries to login to a secure application with an incorrect password, and submits an incorrect password in more than three consecutive attempts, the user may be locked out of the application.

- In Web applications, processes often need to remember their state. If a user of a shopping-cart application tries to submit his order without having entered his billing information, his error must be reported to him and the *submit* event must be rejected.

- In parsing a programming language, whether or not a particular character such as a tilde or a question mark is acceptable may depend on where in the source text it occurs. It might be acceptable in a comment, or when enclosed in quote marks, but not otherwise.

One type of event-driven application where issues of state are very visible is in SAX processing of XML. Sax parsers provide startElement and endElement methods that are, in effect, startState and endState methods. When you encounter a start tag of a certain type (e.g. "<h1>"), you are in effect entering a state. All succeeding elements in the document occur within the context of that state until you encounter the corresponding end tag (i.e. "</h1>").

Here is a snippet of SAX processing code (in Python) that checks to make sure that the input doesn't contain any "<br>" tags or heading tags nested inside other heading tags.

```
...
# the CustomizedHandler class extends the SAX parser's ContentHandler class
class CustomizedHandler(ContentHandler):

   VALID_HEADING_ELEMENTS = ["h1", "h2", "h3", "h4", "h5"]
   inHeadingStatus = False
   currentHeadingTag = None

   def startElement(self, argTag, argAttrs):

      if argTag == "br":
         if self.inHeadingStatus == True:
            raise InvalidTagError(argTag, self.currentHeadingTag)

      elif argTag in self.VALID_HEADING_ELEMENTS:

         if self.inHeadingStatus == True:
            raise InvalidTagError(argTag, self.currentHeadingTag)
         else:
            self.inHeadingStatus = True
            self.currentHeadingTag = argTag

      else: # tag is not a heading element
         pass


   def endElement(self, argTag):

      if argTag in self.VALID_HEADING_ELEMENTS:

         if self.inHeadingStatus == True:
            self.inHeadingStatus = False
            self.currentHeadingTag = None
         else:
            # actually, the SAX parser would catch this error
            raise InvalidTagError(argTag,self.currentHeadingTag)


      else: # tag is not a heading element
         pass
...
```

Compare the structure of this code to the code that parsed letters and spaces.

The parsing program was quite simple, so that once we were inside the *handleSpace* or *handleLetter* functions, the parsing program knew everything it needed to know about the transaction that it was processing.   That's not enough in this program. Here, once we are inside the *startElement* or *endElement* methods, we need to do further checking to see what kind of tag we're processing.

But once we know what kind of event we're processing, the processing is quite similar. Basically, it involves setting and checking status information (e.g. *inHeadingStatus*) and state-vector information (e.g. *currentHeadingTag*), and accepting or rejecting transactions based on the current state of the application.

Ways to remember state

Many stateful applications need to remember their state only as long as they are actually running. Such applications have no problem remembering their state information – they simply store it in memory. In our parsing program, for instance, we used the *state* and *outstring* global variables. In our SAX content handler, we used the *inHeadingStatus* and *currentHeadingTag* instance variables of the content handler object.

But other stateful applications need to suspend execution, which means that they need to store their state information somewhere other than in memory. Such an application may:

- take responsibility for storing state information on some persistent medium, such as a file or database on disk. It retrieves its state information from the database when it starts up, and it stores (persists) its state information back to the database just before it terminates.

or it may:

- delegate the responsibility for remembering its state information to its caller. It receives its state information from its caller when it starts, and it returns its state information to its caller when it terminates.

Web applications are good examples of this kind of stateful application, because they often use both of these strategies in remembering state information. Consider nozama.com, an imaginary Web shopping application in which a typical user with a Web browser:

- opens the nozama.com Web page
- goes through a selection procedure in which he repeatedly browses the Nozama catalog of products, and adds products, one at a time, to his shopping cart
- enters his billing and shipping information
- submits his order

At any given time, there may be hundreds or thousands of simultaneous users in the middle of the shopping process. Nozama must handle a stream of service requests in which requests are interleaved from different shoppers in different stages of the shopping process. Unfortunately, simple Web technology doesn't supply Nozama with a direct link to any given shopper.

> HTTP is a stateless protocol: it provides no built-in way for a server to recognize a sequence of requests all originating from the same user....
>
> The HTTP state problem can best be understood if you imagine an online chat forum where you are the guest of honor. Picture dozens of chat users, all conversing with you at the same time. They are asking you questions, responding to your questions.... Now imagine that when each participant writes to you, the chat forum doesn't tell you who's speaking! All you see is a bunch of questions and statements mixed in with each other. In this kind of forum, the best you can do is to hold simple conversations, perhaps answering direct questions. If you try to do anything more, such as ask someone a question in return, you won't necessarily know when the answer comes back. This is exactly the HTTP state problem. The HTTP server sees only a series of requests—it needs extra help to know exactly who's making a request.[11]

[11] *Java Servlet Programming* (2nd ed.) by Jason Hunter with William Crawford (O'Reilly, 2001), "Chapter 7: Session Tracking", p. 200.

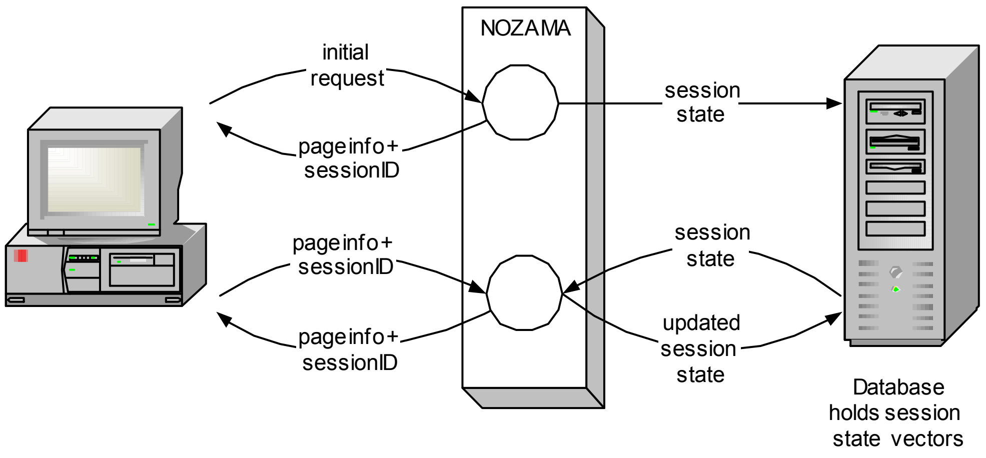
What Nozama and similar applications need is a way to identify a conversation with a particular client. Such a conversation is called a *session*. What Nozama needs is a way to manage the process of creating, maintaining, and remembering session information.

Here's how it does it.

When a user opens the Nozama web site, Nozama creates a temporary *session object*[12]. As the user goes through the shopping process, Nozama adds information that the user supplies– his product choices, his billing and shipping information– to the state information of his session object. Finally, the life of the session ends when the user submits his order.

Nozama tracks sessions by giving each session a *session ID* and by storing the session's state vector (its state information) in a database in which the session ID is the key of the state vector.[13]

Once a session has been started, Nozama includes the session ID in every Web page that it sends to the user, encoded in such a way[14] that it will be sent back to Nozama as part of every page request by the user. When Nozama receives a page request containing a session ID, it uses the session ID as a key to retrieve the session state vector information from the database. After Nozama has processed the user's request, it updates and replaces the state vector in the database, and returns a response to the user.



[12] Other kinds of applications may require a user to go through an explicit *login* process, in which the user supplies user ID and a password, before creating a session object.

[13] State vector information for a session typically consists of one *context,* but may consist of more than one.

[14] There are a number of ways in which this can be done, including coding the session ID in a hidden form field of the Web page, URL rewriting, and using *session cookies*. A full discussion of these techniques is beyond the scope of this paper.

Remember that we said that a stateful application could remember its state information by storing the information on some persistent medium, or by delegating responsibility for remembering the information to its caller. Nozama uses both of these strategies. It remembers the actual state information by storing it on disk, in a database. But it also gives the caller – in this case, the client's browser – the responsibility of remembering one piece of state information: the session ID.

An alternative strategy would be for Nozama to pass the entire session state back and forth to the browser, and let the browser remember it. If Nozama did this, it wouldn't need to use a database to store session information, and it could be much simpler. But transmitting potentially large amounts of state vector information back and forth to/from the client's browser could slow down response time considerably.

The alternative strategy does have one distinct advantage. If the user aborts the shopping process before completion, it leaves no orphaned session information in the database. In contrast, with the database strategy, Nozama must implement the notion of a *session time-out* to detect aborted sessions. When a session's state information has not been accessed after a certain amount of time (say 30 minutes), Nozama will consider the session to have been aborted, and must delete the session state information from the database.

Conclusion

This concludes our brief introduction to event-driven programming — really, to the Handlers pattern and its variants – and related programming issues.

As you can see, understanding event-driven programming is the key to being able to perform many software development tasks: object-oriented programming, object-oriented systems analysis and design, parsing XML with a SAX parser, GUI programming, Web programming, and even lexing and parsing.

Good luck with your event-driven programming!

Appendix A – Abstract methods in Python

In Python 2.4 and later, it is possible to use a decorator to create abstract methods.

```
# define a decorator function for abstract methods
def abstractmethod(f):
    methodName = f.__name__
    def temp(self, *args, **kwargs):
        raise NotImplementedError(
            "Attempt to invoke unimplemented abstract method %s"
            % methodName)
    return temp


class TestClass: # an abstract class, because it contains an abstract method
    @abstractmethod
    def TestMethod(self): pass

t = TestClass() # create an instance of the abstract class
t.TestMethod()  # invocation of the abstract method raises an exception
```

For a more sophisticated approach to creating abstract methods in Python, see:
http://www.lychnis.net/blosxom/programming/python-abstract-methods-3.lychnis

Appendix B – SAX parsing in Python

```
"""Use a SAX parser to read in an XML file and write it out again."""

import sys, os
import xml.sax
from xml.sax.handler import ContentHandler
from xml.sax.saxutils   import escape

class CustomizedHandler(ContentHandler):

        def setOutfileName(self, argOutfileName):
                # Remember the output file so we can write to it.
                self.OutfileName = argOutfileName
                self.Outfile     = open(self.OutfileName, "w")

        def closeOutfile(self):
                self.Outfile.close()

        def write(self, argString):
                self.Outfile.write(argString)

        def startDocument(self):
                pass

        def endDocument(self):
                pass

        def setDocumentLocator(self, argLocator):
                self.myDocumentLocator = argLocator

        def startElement(self, argTag, argAttrs):
                # argAttrs is a list of tuples.
                # Each tuple is a pair of (attribute_name, attribute_value)

                attributes = ""
                for name in argAttrs.getNames():
                        value = argAttrs.getValue(name)
                        attributes = attributes+(' %s="%s"' % (name, value))
                self.Outfile.write("<%s%s>" % (argTag, attributes))


        def endElement(self, argTag):
                self.write("</%s>" % argTag)

        def characters(self, argString):
                self.write(escape(argString))


        def ignorableWhitespace(self, argString):
                self.write(argString)

        def skippedEntity(self, argString):
                self.write("&%s;" % argString)

        def handleDecl(self, argString):
                self.write("<!%s>" % argString)

        def processingInstruction(self, argString):
                # handle a processing instruction
                self.write("<?%s>" % argString)
```

```
def main(myInfileName, myOutfileName ):

       myContentHandler = CustomizedHandler()
       myParser = xml.sax.make_parser()
       myParser.setContentHandler(myContentHandler)
       myContentHandler.setOutfileName(myOutfileName)

       myInfile = open(myInfileName, "r")  # open the input file
       myParser.parse(myInfile)            # parse it

       myInfile.close()                    # close the input file
       myContentHandler.closeOutfile()     # close the output file


def dq(s): # Enclose a string argument in double quotes
       return '"'+ s + '"'

if __name__ == "__main__":
       print "Starting SaxParserDemo"
       infileName = "SaxParserDemo_in.txt"
       outfileName = "SaxParserDemo_out.txt"

       # -------- create an input file to test our program -------------
       infile = open(infileName, "w")
       infile.write("""<html><head>
<style><!-- This is comment --></style>
</head><body>This is an ampersand: &amp;
<br/>This is a gt sign: &gt; and an lt sign: &lt;
<![CDATA[an ampersand & and gt sign > and lt sign < ]]>
</body></html>""")
       infile.close()

       main(infileName, outfileName)  # call main() to process the test file
       print "Ending   SaxParserDemo"
```