DigiPen
INSTITUTE OF TECHNOLOGY

Lecture 5

Procedural vs Event Driven Programming

CSD1130
Game
Implementation
Techniques

Event Driven Programming

1. Describe procedural (or structured or modular) programming (concept by Edgar Dijkstra).
   a. Procedural programming uses a natural approach to problem solving – specify a sequence of steps telling the computer what to do. The flow of program execution proceeds from the first step to the next until the last step is reached and the program terminates *(Figure 1)*.

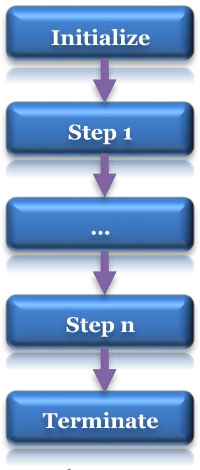


Figure 1: Procedural Programming

   b. Consider the task of writing a program to access a description of shapes that were stored in a database and then display them. Consider the natural approach to solving the above problem in terms of the steps required:
      i. Locate the list of shapes in the database.
      ii. Open the list of shapes.
      iii. Sort the list according to some rules.
      iv. Display the individual shapes on the computer screen.

   c. Each one of the four steps can be broken down further into the steps required to implement it. Consider step 4. For each shape in the list, do the following:
      i. Identify the type of shape.
      ii. Get the location of the shape.
      iii. Call the appropriate function that will display the shape, giving it the shape's location.

d. In procedural programming, the problem is decomposed into the functional steps that comprise it because it is easier to deal with smaller pieces than it is to deal with the problem in its entirety. It is like the approach used by humans in cooking a dish by following its recipe, manufacturing a car or even making a game. One consequence of procedural programming is that there is one "main" program that is responsible for controlling its subprograms. It is a natural outcome of decomposing functions into smaller functions *(Figure 4)*. The program's flow can also be subject to branching *(Figure 2)* and looping *(Figure 3)*.

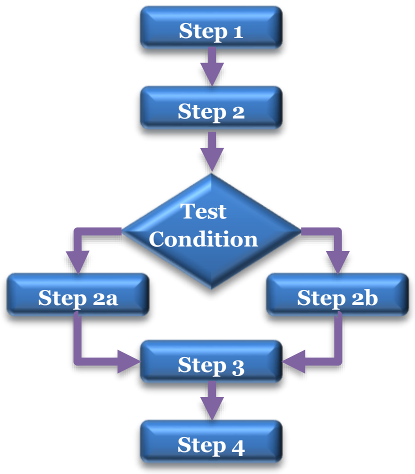


Figure 2:
Branching

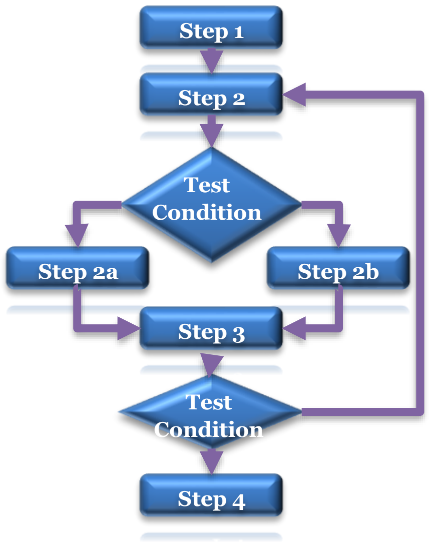


Figure 3:
Looping

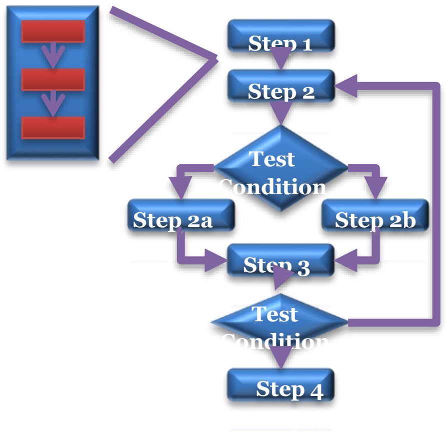


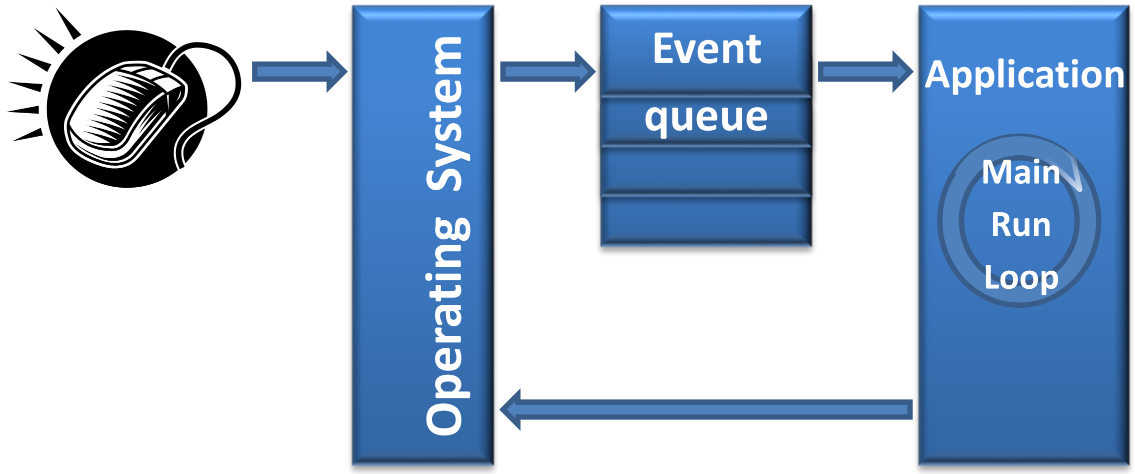
Figure 4:
Functional Decomposition

e. By adding user input to the application, it makes interactive. Nevertheless, the application is still responsible for controlling its subprograms and not the user. This type of applications is called control-driven applications where its main advantages are straight development and the interactions between users and programs can be easily modeled. However, this leads to wasting system resources (polling) by continuous checking for the user input (example: CS120 programming assignments). Also, complex interfaces and asynchronous interactions cannot be implemented.

- Example:
  If an application is waiting for a key press and a mouse is clicked, the mouse click is ignored (e.g., FPS games)

```
while (!quit)
{
      Prompt the user
      Read input from keyboard or mouse
      Parse user input to determine user choice or action
      Generate output
      Write output
}
```

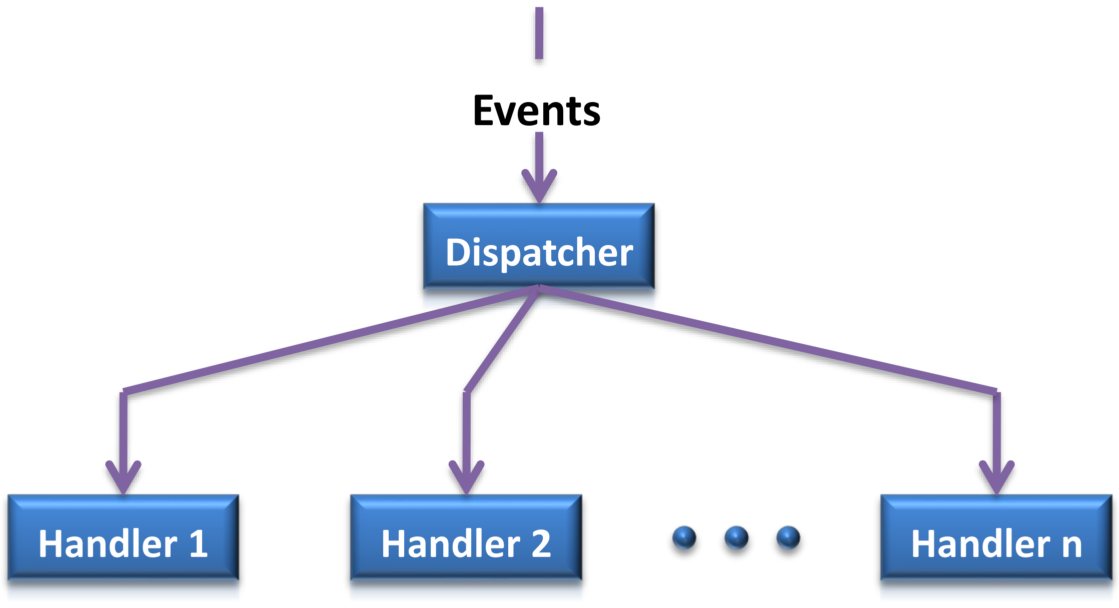
2. In contrast, in a typical GUI-based application, most of the action after initialization occurs in response to user actions such as mouse click, mouse movement, menu option selection, or text insertion. Such actions trigger *events* (defined as a type of signal to the program that something has happened) and special methods in the application called *event handlers* are invoked. When no events are generated, the application does nothing. In other words, the user is in control of the application or so called the Hollywood principle "Don't call us, we'll call you".



3. The most fundamental difference between procedural and event-driven programming is that, in event-driven programming, other software, or the operating system itself, is calling event handlers in

the application. Instructions are not executed sequentially from first to last as in procedural programs. Instead, in event-driven programs, the application does not know which functions are invoked. The application can decide which events to handle – but the application cannot know in advance the exact order in which those events will occur.

4. Event-based applications can be described using the *Handlers pattern*.



b. A stream of data items is called *events*.

c. The job of the *dispatcher* is to take each item 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 inputs, 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. If the dispatcher was a human being, then I would say he would be the mailman, the events are the mail, and the handlers are the mail receivers. The mailman gets the mail, on which it has a specific address, and delivers the mail to its appropriate location. Once the mail is delivered and opened by the receivers, they will act upon the content of the mail.

d. For some applications (especially those controlling hardware), the input stream is infinite. For most event-handling applications, the event stream is finite with an end indicated by a special event: ESCAPE key press, left-click on CLOSE button of GUI, EOF marker in file reader. In such 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.

e. *Handlers* perform the appropriate action specified by the event type.

5. Here is pseudo-code for a typical dispatcher showing an event loop, the quit operation, the determination of the event type and the selection of the appropriate handler based on that type, and the treatment of events without any handlers.

```
while (1) // do forever
{
        get event from input stream
        if( event type == EndOfEventStream)
                Quit() //Break out of event loop
        elif( event type == EventTypeZero)
                ExecuteEventTypeZero(event information);
        elif( event type == EventTypeOne)
                ExecuteEventTypeOne(event information);
        else    // handle unrecognized event type
                // ignore the event or raise an exception
}
```