Inf-1400 - Objektorientert Programmering

Event-driven programming

John Markus Bjørndalen

Department of Computer Science
University of Tromsø
Norway

2017-03-28

Programming styles

Programming style so far in this course might be described as batch-oriented:

- Program flows in a preplanned way
- It may wait for input from the user

Event-driven programming:

- Program decides what to do next based on events that occur (mouse clicks, keyboard clicks, timers)
- Graphical user interface (GUI) often programmed using event-driven programming

Event-driven programming

Not a part of object-oriented programming, but:

- Useful way to write many programs (GUI, ...)
- Objects are useful for modeling events and event handlers

Basics of event-driven programming

Our PyGame main loops use event-driven programming:

```
while not finished:
    for event in pygame.event.get():
        if event.type == QUIT:
            finished = True
    ...
```

`pygame.event.get()` fetches all events that have happened since the last call. Returns a sequence of events.

Event queues

`pygame.event.get()` transfers control over to PyGame:

- Does bookkeeping and detects new events
- Stores new events on an event queue (temporarily)
- Removes and returns the events as a sequence

Event handling

Determining the type of event and responding with appropriate action is called event handling:

```
for event in pygame.event.get():
    if event.type == QUIT:
        finished = True
```

Event handling

Events can also be handled by event-handlers implemented using functions or objects.

Determining which handler to call is called dispatching:

```
for event in pygame.event.get():
    if event.type == FOO:
        # event handler function
        foo_handler(event)
    if event.type == BAR:
        # calling handle()-method on object
        bar_handler.handle(event)
```

Event dispatchers

Routes events to event handlers. Can be implemented as an object. Simple example:

```
class Dispatcher:
    def __init__(self):
        self.__handlers = {}

    def register_handler(self, etype, handler):
        # NB: This simple code assumes only a single
        # handler for each type
        self.__handlers[etype] = handler
    def dispatch(self, event):
        if event.type in self.__handlers:
            self.__handlers[event.type](event)
        else:
            print("Unknown event type", event.type, \
                  pygame.event.event_name(event.type))
```

Event dispatchers

Using it with our PyGame code:

```
def mouse_motion_handler(event):
    print("Moving mouse", event.pos, event.rel)

dispatcher = Dispatcher()
dispatcher.register_handler(MOUSEMOTION,
                            mouse_motion_handler)

finished = False
while not finished:
    for event in pygame.event.get():
        if event.type == QUIT:
            finished = True
        else:
            dispatcher.dispatch(event)
```

Why event dispatchers and handlers?

Extensible concept: users can extend program by adding handlers to dispatcher.

No need to modifiy event dispatcher source code to add new event types.

Generating events

- Event-based systems often provide a method for adding events to the event queue.
- This allows users to add new event types to programs as well as handlers.
- NB: Can also be used for scripted testing.
- In PyGame, this is done using `pygame.event.post`.

Common event types in PyGame

Mouse events

- `event.pos` : xy-position that the mouse moved to
- `event.rel` : relative motion since last event
- `event.buttons`: state of buttons when the event was created

Keyboard events:

- KeyDown (when a key is pressed) and KeyUp (when the key is released).
- The event contains information about the key pressed and modifiers (shift, alt etc).

Timer events

Timers are events that are set to fire off after a certain amount of time.
Can be one-shot or periodic.
PyGame timers are periodic and fire off a an event of a specified type every N miliseconds.

```
# fires an event of type USEREVENT+1 every
# 2000 miliseconds.
pygame.time.set_timer(USEREVENT+1, 2000)
```

Example use: periodic update of information, such as the arms of a clock.

Event-driven programming problems

Allowing handlers to change state: requires access to the state. (Example: variable in object or method).
Some solutions:

- Global variables
- Default params to functions
- Complex event handler objects

How do you reason about the program?

- What happens when and in which order?
- When did somebody change this state?

Links

- Pygame events :
  http://www.pygame.org/docs/ref/event.html
- **http://en.wikipedia.org/wiki/Event-driven_programming**