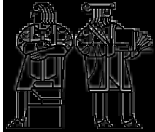


6.170 Lecture 18 Patterns and events



John Chapin
MIT EECS
3-14-2000

Outline

Design patterns
Observer
Event-driven programming

Design pattern

A piece of a design intended to be reused
design reuse, rather than code reuse

Contains:
design descriptions: MDD, OM, others as needed
interfaces

motivation	what problem justifies this pattern?
example	how to use the pattern?
applicability	where is this pattern useful?
benefits	what does it buy you?
potential problems	what does it cost you?

Example pattern: Observer

Motivation: Common UI problem
Show multiple views of underlying data
Views are correlated

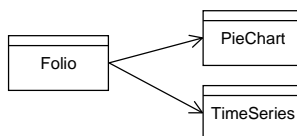
Example:

stock portfolio	Folio
time series graph	TimeSeries
investment allocation	PieChart

when stock value in Folio changes, both TimeSeries
and PieChart should update their views.

Obvious control strategy (poor)

Folio
stores pointers to PieChart and TimeSeries
tells them what data to display when a change occurs



Simple, easy to understand
Folio must know what data to display in PieChart
Adding new displays complicates Folio

Observer pattern (better)

Folio does not have to push data to the views
Instead, the views pull data from Folio

Folio (the subject)
stores list of Observers
notifies each Observer when a change occurs

PieChart and TimeSeries (the Observers)
when notified, call Folio observer methods to get data

Detailed example

```

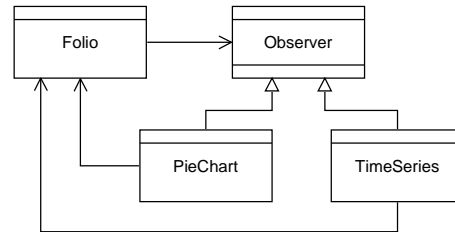
interface Observer {
    void update ();
}

class Folio {
    // subject operations for Observer pattern
    public void attach (Observer o);
    public void detach (Observer o);
    protected void notify ();
    // observers: getPrice(string), getTotalValue()
    // mutators: buy(string, price, numShares)
}

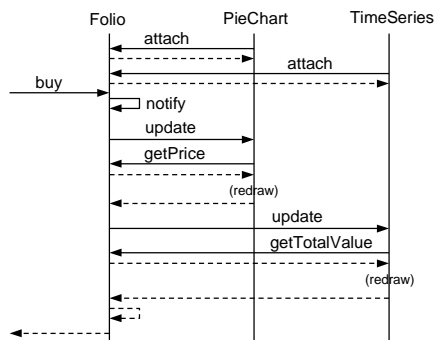
class PieChart implements Observer {
    // Observer operations for Observer pattern
    void update ();
    // creators, observers, mutators not shown
}

```

MDD



Message sequence chart



Applicability and benefits

Use the Observer pattern when

- multiple views of data must be kept correlated
- want to add and remove views easily

A "view" might be something internal, not UI
OK if views mutate the subject

Benefits

- Each observer completely local
- what information is displayed
- how it is shown
- Can add a new observer
- without changing any code in subject or observers

Potential problems

- Updates may be unexpectedly expensive
- Confusing control flow if update() has side effects
- Cascades of updates
- update() called while observer state inconsistent
- if some other object it calls mutates the subject

Variants of the pattern help solve these problems
see the book

Summary so far

Notion of a design pattern**Observer**

will be useful in final project



Events: The issue

Challenge: how to do several things at the same time

- Operating system: handle requests from many apps
- Server: process queries from multiple clients
- GizmoBall: animate ball AND respond to user input

Should you use threads?

Claims:

- For most purposes, events are better.
- Use threads only when true CPU concurrency is needed.



What are threads?

State of the program
(objects & static variables)



multiple threads
operate simultaneously
on the program state

General-purpose solution for managing concurrency.

Characterized by:

- Multiple independent execution streams.
- Shared state.
- Synchronization (e.g. locks, conditions).



What are threads used for?

Operating systems:

- one kernel thread for each user process.

Servers:

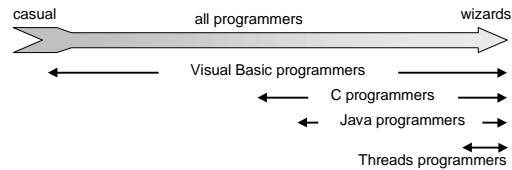
- one thread per request (to overlap I/Os).

GUIs:

- one thread for main(), one thread for Swing
so display doesn't freeze when computing
video, animations
- easier to deal with multiple rates in one program



What's wrong with threads?



Too hard for most programmers to use.

Even for experts, development is painful.



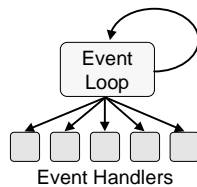
Event-driven programming

One execution stream:
no CPU concurrency.

Event loop waits for events,
invokes handler based on
type of event.

No preemption.

Handlers generally short-lived.



Example uses of events.

GUI:

- One handler for each event
press button, invoke menu entry, etc.
- Handler implements behavior
undo, delete gizmo, etc.

Server:

- One handler for each source of input (socket, etc.).
- Handler processes incoming request, sends
response.
- Use nonblocking I/O for I/O overlap.



Problems with events (1)

Long-running handlers cause application to freeze.

Break up handlers

1. Do part of computation
2. store partial result
3. return to event loop
4. finish later in another handler

Use nonblocking I/O

Periodically call event loop from within handler
adds complexity

Create subprocesses for independent work



Problems with events (2)

Must store local state in heap between events

```
void handler() {
    int i = this.stored_i;
    while (i < 1000) {
        do_work(i);
        i++;
        if (time_up()) { stored_i = i; return; }
    }
    stored_i = 0;
}
```

No CPU concurrency (not suitable for scientific apps).

Non-blocking I/O not well supported on standard OS
no non-blocking write in standard Java I/O library



GizmoBall recommendation

Use main() thread to do initialization

In particular, create a window

Let main() return

the main() thread will block for the rest of the run

Execute GizmoBall entirely in the GUI event handlers

When done, call System.exit()

If you do this, no need to worry about thread problems.