##Game Programming Patterns - revisited - Observer ##

* underlies the Model-View-Controller architecture
  ↳ Java.util.Observer & C# event
* ex. unlocking achievements & keeping achievements out of the rest of our codebase to avoid spaghetti

"[Observer] lets one piece of code announce that something interesting happened w/o actually caring who receives the notification

* the rest of the code still needs to know which notification to send, "we're trying to make systems better, not perfect"
* then the ~~consumer~~ observer of the notification can do other relevant checks to see if the achievement got triggered
* it's nice to be able to rip out or change ~~consumer~~ observer w/out modifying other code.

① Observer
  onNotify
    ↳ assume no modification to Observer[]

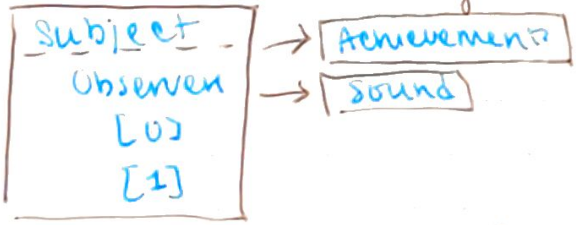
② Subject
  Observer[]
  addObserver
  removeObserver
  Protected notify()

③ hook into code. ex
  Physics extends Subject
  ↳ or has a subject

* each observer needs to be independent of each other

"observer" system, observe the thing that did something interesting (extends subject)

"event" system, observe an obj that represents the interesting thing that happened (have a subject)



* "Too slow", it's walking a list, it's ok
* "Too fast", the observer pattern is synchronous so it will wait until all observers return

"Stay off the UI thread", you need to either return quickly or push slow work to another thread or work queue BEWARY OF DEAD LOCKS THOUGH.

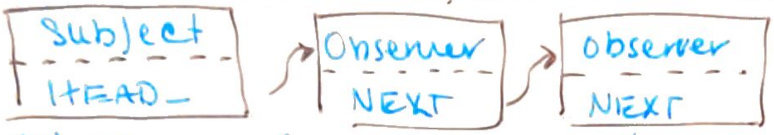
* "too much dynamic allocation"
  ↳ need to avoid fragmentation (obj pool)
  ↳ usually observers are a dynamic list & will need to be garbage collected. Build the list at the beginning & it shouldn't be a big deal.

subj. observer
1 — ∞

$\boxed{\text{Linked observers}}$ (to avoid dynamic allocation)

"If we are willing to put a bit of state in Observer we can [thread] the subject's list through the observers themselves"

*can also make subject a friend class

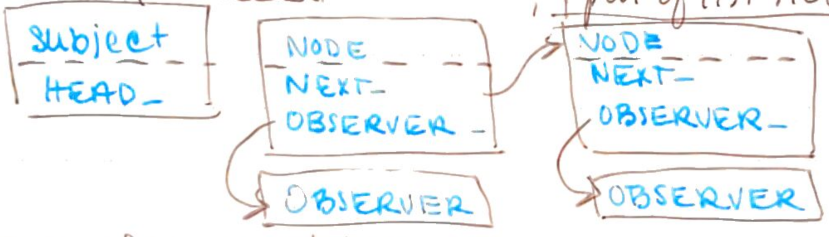


*Subject still has notify
add Observer + remove Observer

"It's a tenet of good observer discipline that two observers observing the same subject should have no ordering dependencies relative to each other" *If order does matter then that coupling could bite you

→ with this set up (linked list) you can now only have an observer observing one subject which is commonly ok, but might not be.

When you need subj. observer ∞ · ∞, $\boxed{\text{pool of list nodes}}$ + No dynamic allocation



*∞ nodes can point to the same observer can observe ∞ subjects simultaneously } then pre-allocate a pool of them

Types of linked lists
① a node object contains the data
② intrusive linked list where the data (aka observer) contained the node (aka NEXT_ pointer)

"The reason design patterns get a bad rap is because people apply good patterns to the wrong problem + end up making things worse"

*Technical problem: destroying subjects + observers
⚠avoid dangling pointers
① add removeObserver() call to observer's destructor + leave it with them to clean themselves up (requires an observer to know which subjects it's observing)
② have subject send a "dying" notification so observers can respond + clean up
③ make base observer class unregister when it gets destroyed, then you need to build in a list of subjects it's observing

lapsed listener problem

↳ when subjects retain references to their listeners & you have zombie UI objects lingering in memory.

"Be disciplined about unregistration"

Wider Problem, what's going on?

* the observer pattern helps loosen the coupling b/w code
* if a bug presents itself + it chains across observers & if the observers are in a list you can only catch the issue at runtime

"If you often need to think about both sides of some communication in order to understand a part of the program don't use the Observer pattern ∴ prefer something more explicit."

* this pattern is good for unrelated lumps of code but not as useful within a single lump of code dedicated to one feature or aspect

Modern Times

* now an "observer" is likely to be a reference to a function
  ex C# "events" register "delegates" (reference to a method)
  Javascript, observers can implement EventListener protocol OR (more commonly) be a function
  * prefer to register member function pointers as observers vs instances of an interface

Observers Tomorrow

* in large apps a lot of code looks the same
  ① get notified state changed
  ② imperatively modify a chunk of UI to reflect new state
* people have been trying to solve this tedium for a while ("dataflow" or "functional reactive" programming)
* Recently people have been using data binding, a little intensive + covers the busywork of tweaking a UI element or calculated property