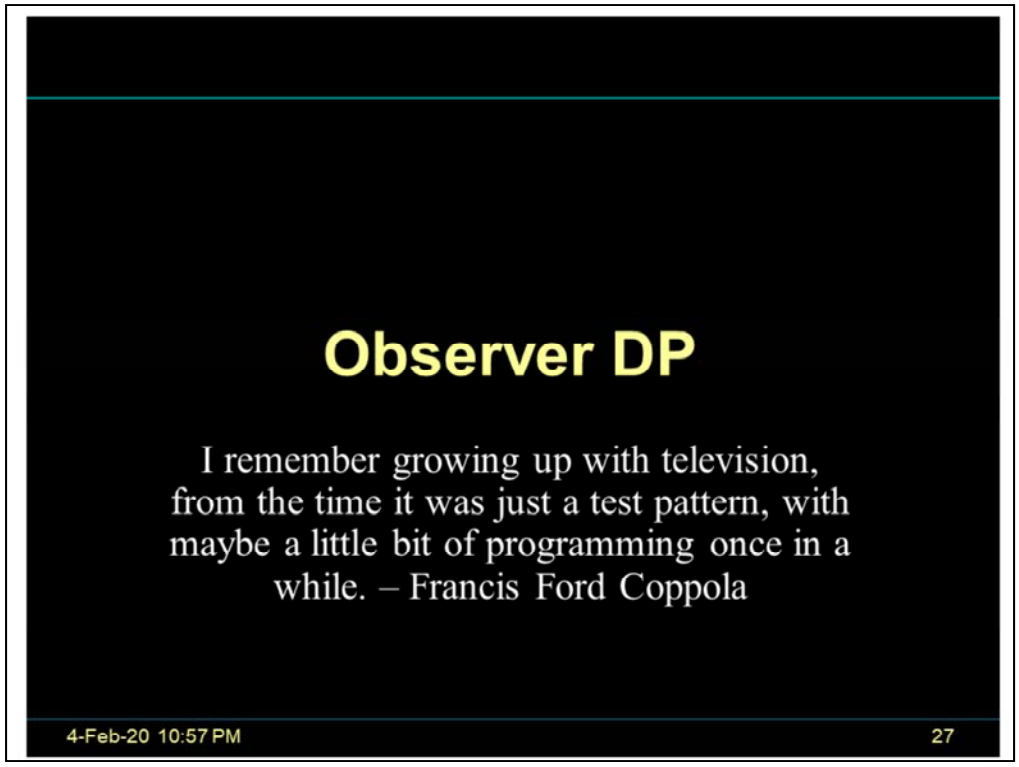
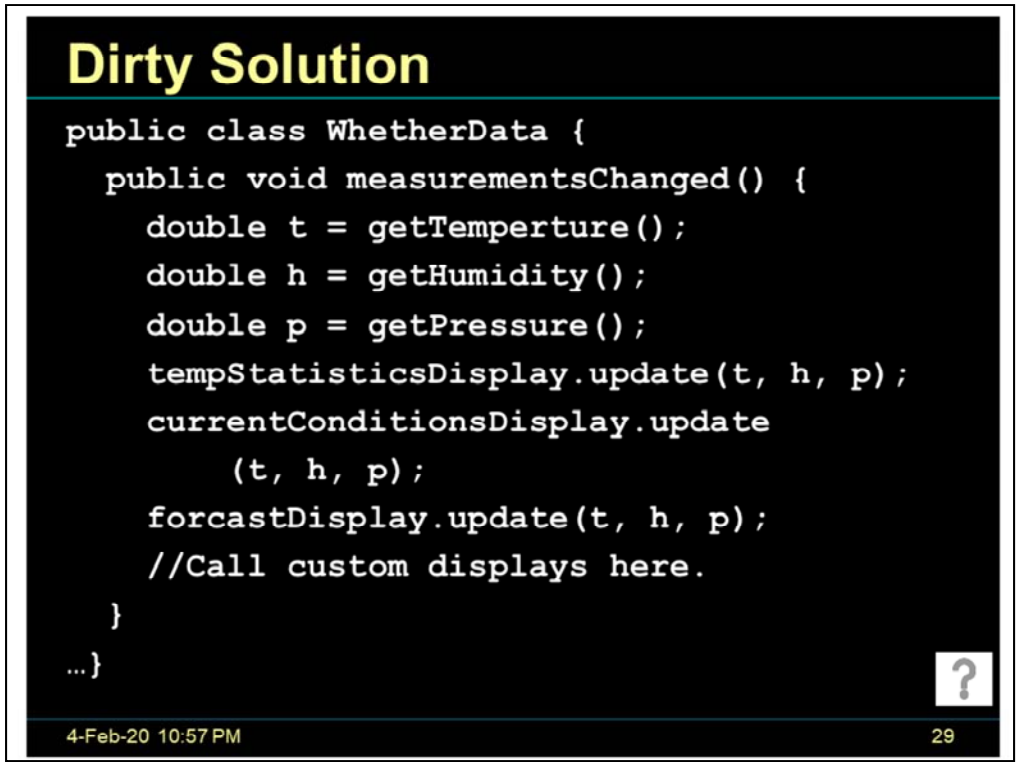

Observer DP
I remember growing up with television, from the time it was just a test pattern, with maybe a little bit of programming once in a while. – Francis Ford Coppola
4-Feb-20 10:57 PM
27

Problem

- Weather information is available via functions:
  - getTemperature, getHumidity and getPressure
- When any whether parameter changes,
  - the measurementsChanged function is called.
  - We need to update the display of
    - Temperature. It shows average temp, max temp, min temp.
    - Current conditions.
    - Forecast.
    - Custom plug-in displays.

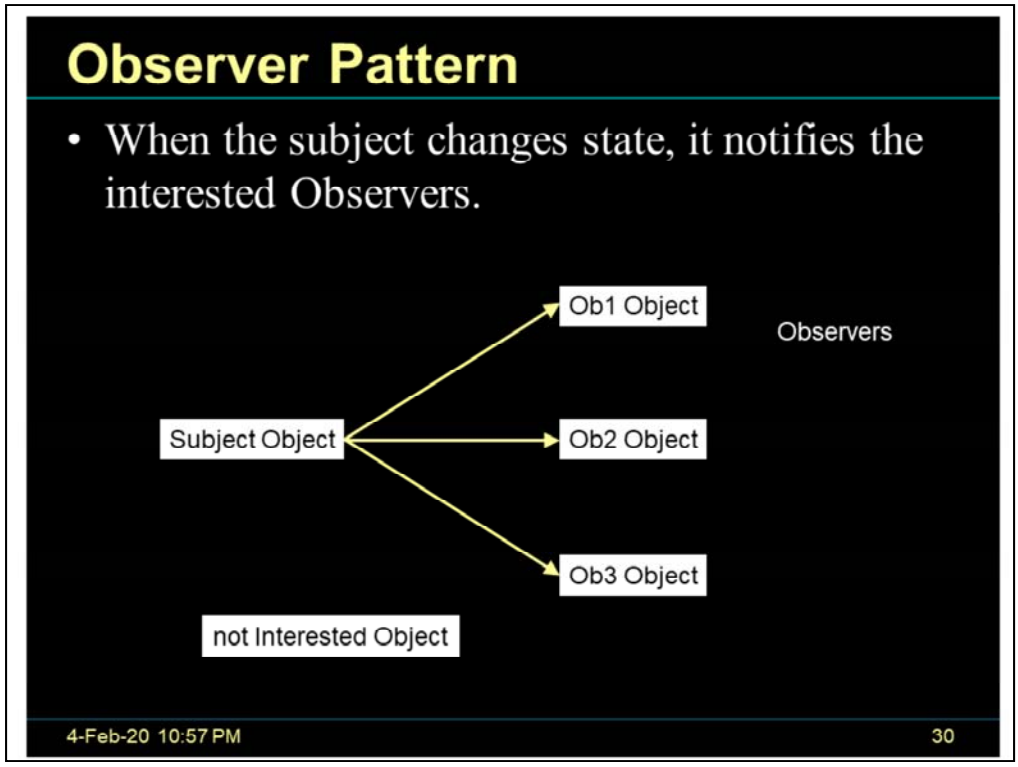


We are coding to implementation and not interfaces.

For every new display, we intend to alter code.

We have not encapsulated the portion that changes.

We have no way of adding/removing display elements at run time.

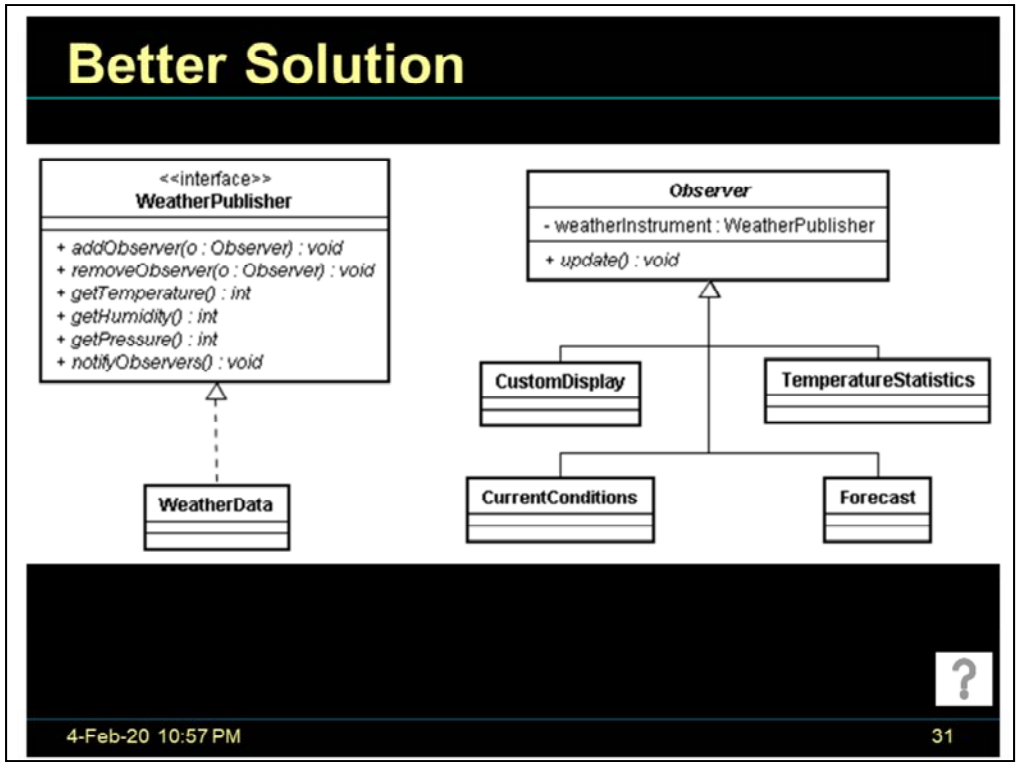


Strive for loose coupling between objects that interact.

A subject knows its observers.

Any number of observers may know the subject.

The observer has an updating interface, so that the subject can notify it about changes.



Publishers + Subscribers = Observer Pattern.

The observer pattern defines one-to-many relationship between objects.

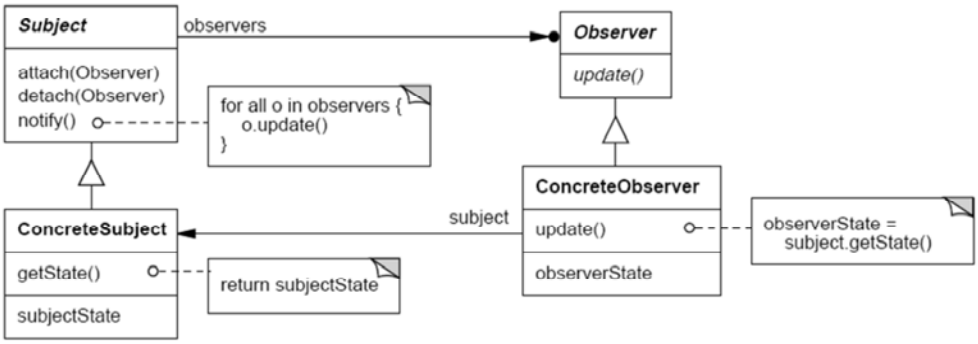
When one object changes state, all its dependents objects are notified.

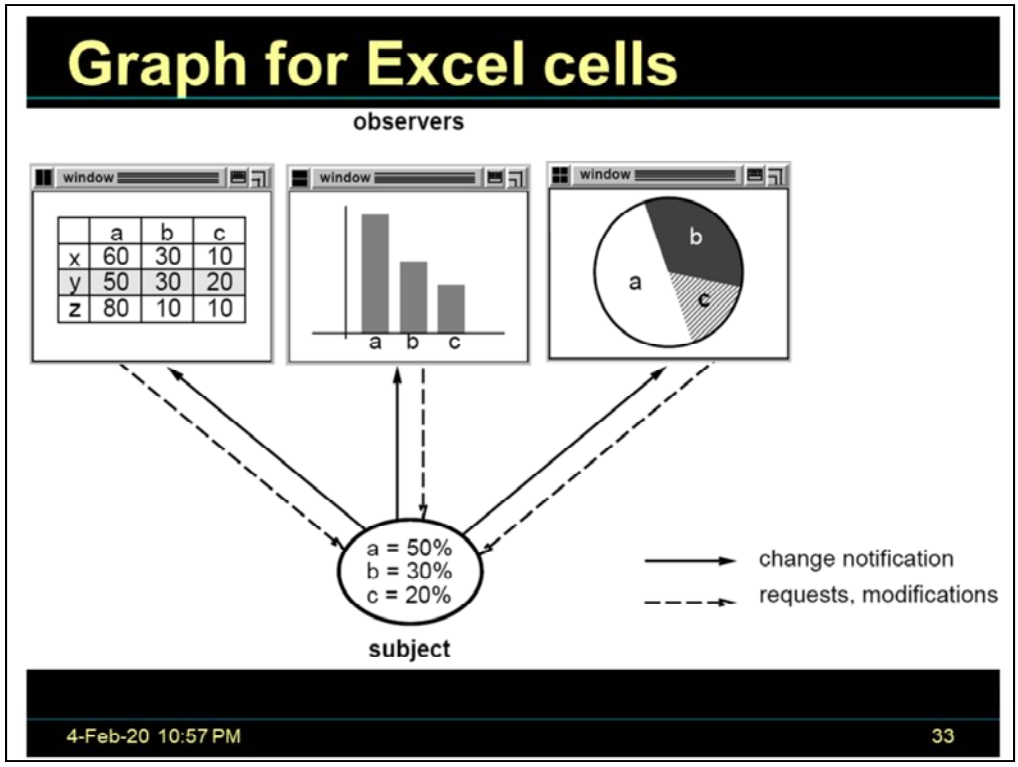
Subject is one. Observers are many.

Subject and Observers are loosely coupled.

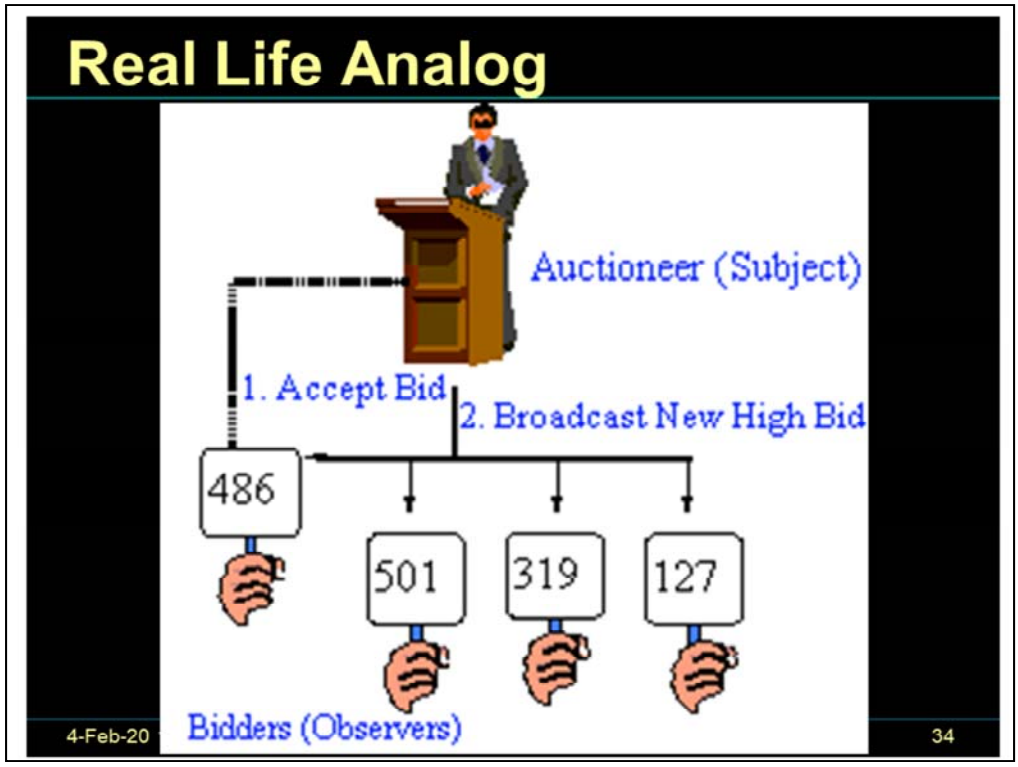
Code examples

Observer DP


Subject
observers
attach(Observer)
detach(Observer)
notify()
for all o in observers {
o.update()
}
Observer
update()
ConcreteObserver
update()
observerState
observerState =
subject.getState()
subject
ConcreteSubject
getState()
subjectState
return subjectState



When we change a value in a cell of an excel spreadsheet, the graphs, pivot tables, etc. should reflect the change.

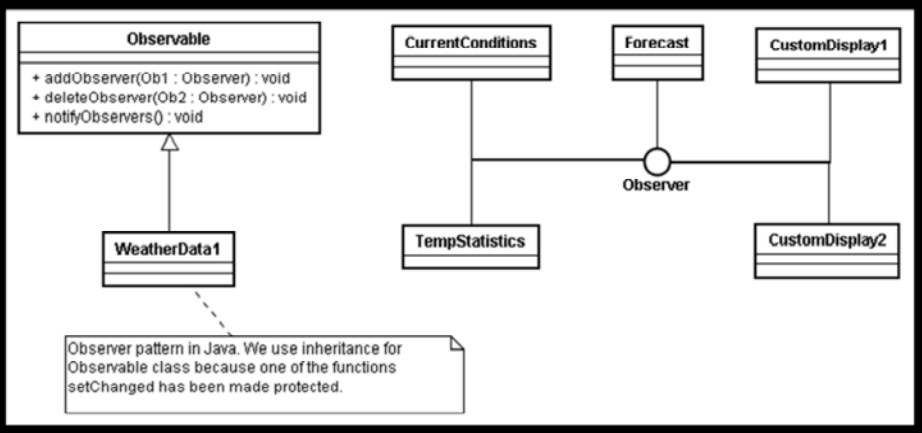


The *Observer* defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.

Uses of Observer pattern

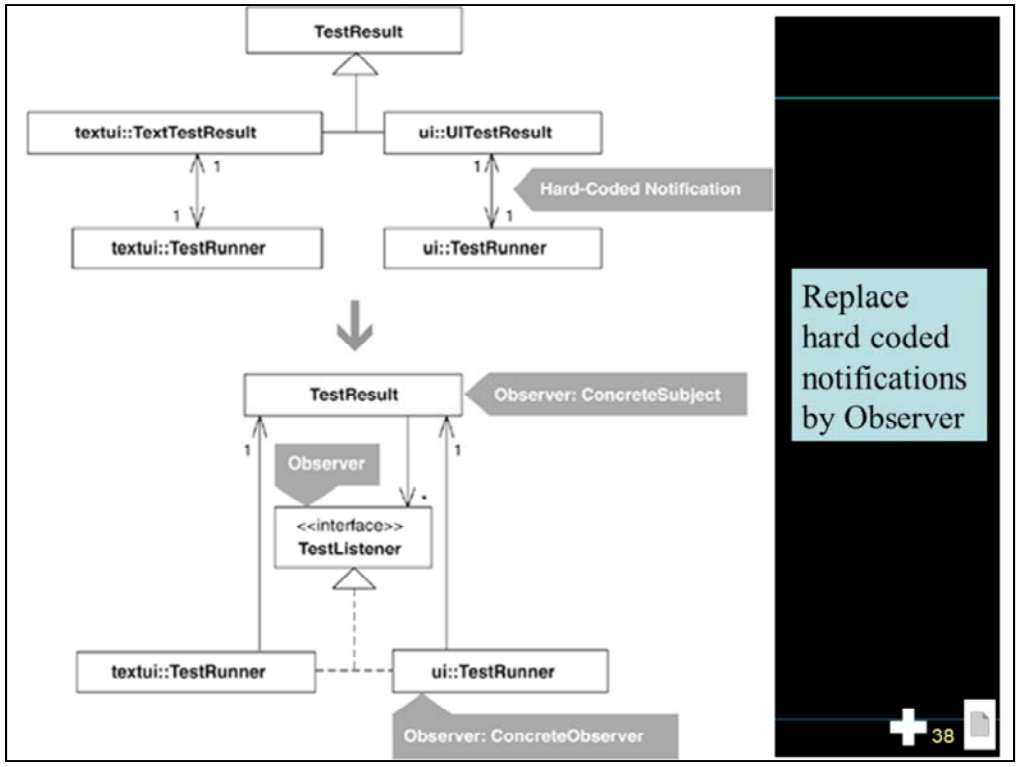
- The Observer pattern is used heavily in JDK
  - In a GUI framework like Swing, we listen for events. The observer pattern is normally used to notify the interested parties about the events.
- On application exit, we want some processing to be done. Multiple exit Listeners can exist.
- In stock market, when a price reaches some level, we want certain actions to be taken.

Using Observer DP in Java



Examples

- When a customer is added,
  - We need to send a welcome email
  - We need to verify the customer's address
- Also in a rich-client, the GUI thread (EDT) is separate from the Model thread.
  - The Observer can decouple the GUI logic from business Logic
- Real-life Observer Patterns
  - A radio station is broadcasting. Only the radios that are tuned in listen
  - A newspaper is publishing. Subscribers are getting the paper



Subclasses are hard-coded to notify a single instance of another class.

Remove the subclasses by making their superclass capable of notifying one or more instances of any class that implements an Observer interface.

Benefits and Liabilities

+ Loosely couples a subject with its observers.

+ Supports one or many observers.

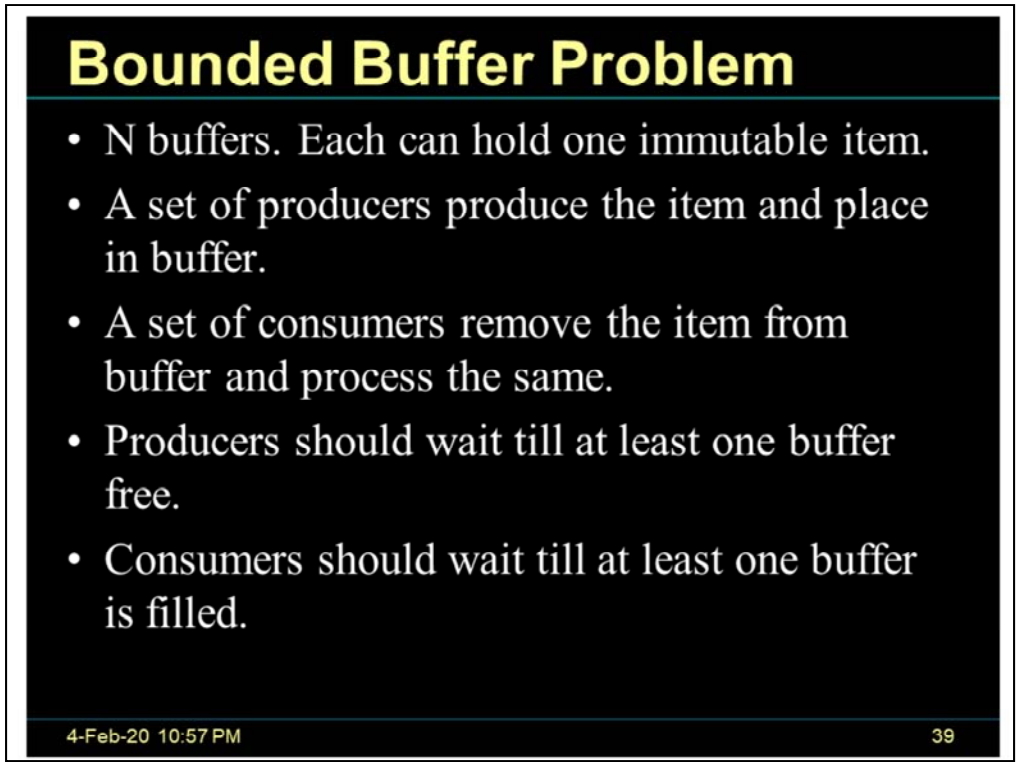
– Complicates a design when a hard-coded notification will suffice.

– Complicates a design when you have cascading notifications.

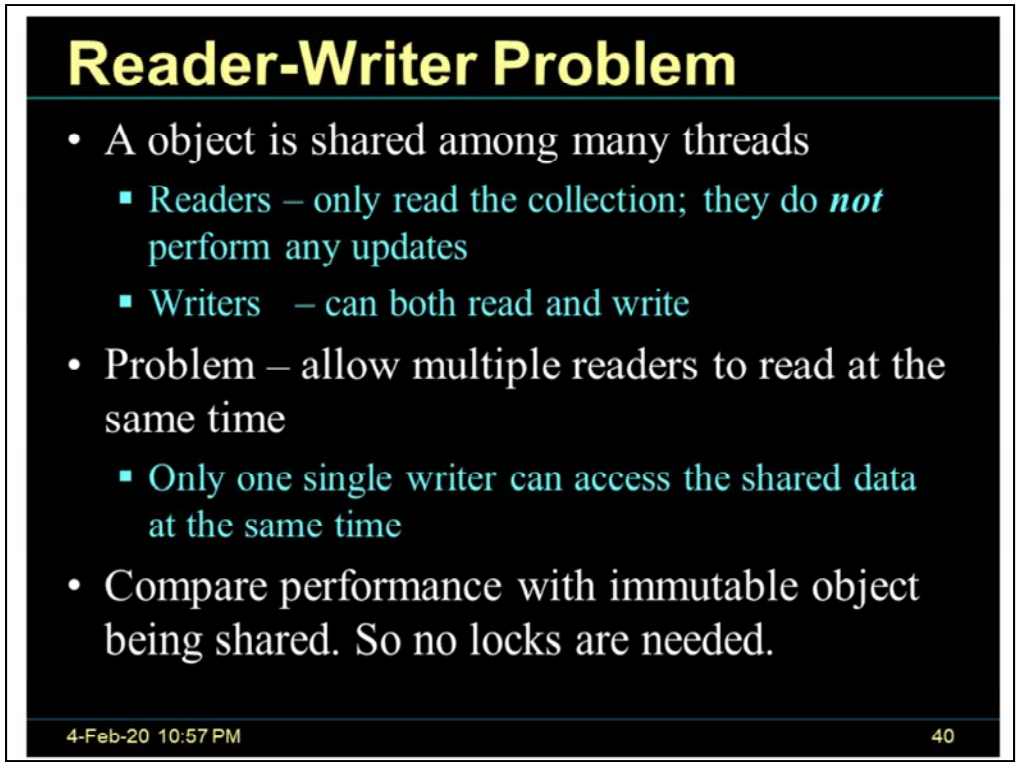
– May cause memory leaks when observers aren't removed from their subjects.

==========

OOAD slides: Polymorphism, Dependency Inversion Principle, DIP implemented, Which is better, Service can be Abstract Now,



Code this problem in Java. Assume a set of 3 threads produce ten million integers. A set of 2 threads consume these integers. Output should be how many evens and how many odds.



Shared Data

Data set

Semaphore **`rw_mutex`** initialized to 1

Semaphore **`mutex`** initialized to 1

Integer **`read_count`** initialized to 0

The structure of a writer process

```
do {
   wait(rw_mutex);
      ...
   /* writing is performed */
      ...
   signal(rw_mutex);
} while (true);
```

The structure of a reader process

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)
```

```
      wait(rw_mutex);
   signal(mutex);
   ...
   /* reading is performed */
   ...
   wait(mutex);
   read count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```