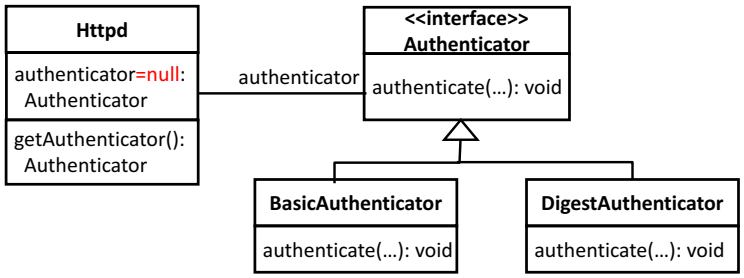
Null Object Design Pattern

Null Object Design Pattern

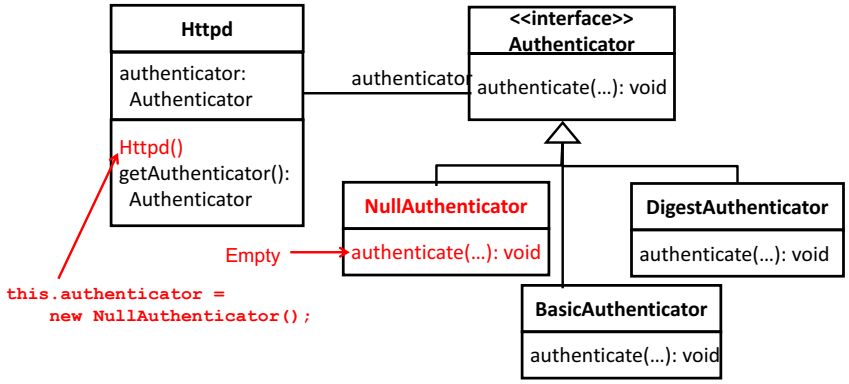
- Intent
  - Encapsulate the implementation decisions of how to do nothing and hide those details from clients
  - Replace a *null-checking* (i.e., if statement) with a neutral/default object that does nothing.

  - B. Woolf, "Null Object," Chapter 1, PLoP 3, Addison-Wesley, 1998.
  - Refactoring: Introduce Null Object
    - http://sourcemaking.com/refactoring/introduce-null-object

An Example: Authentication in HTTP




```
if( getAuthenticator() != null ){
    try{
        getAuthenticator().authenticate(...);
    }catch(AuthenticationFailedException afe){
        makeErrorResponse(afe);
    }
}
makeSuccessfulResponse(...);
```

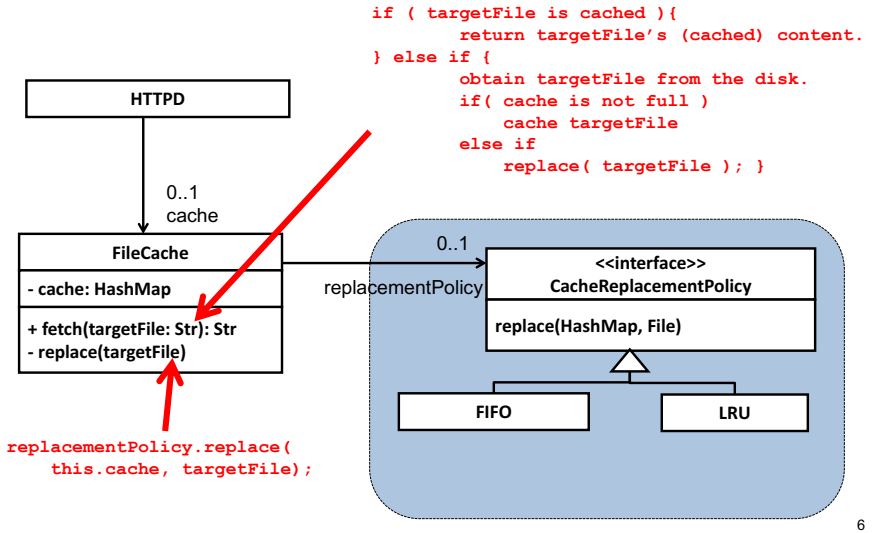



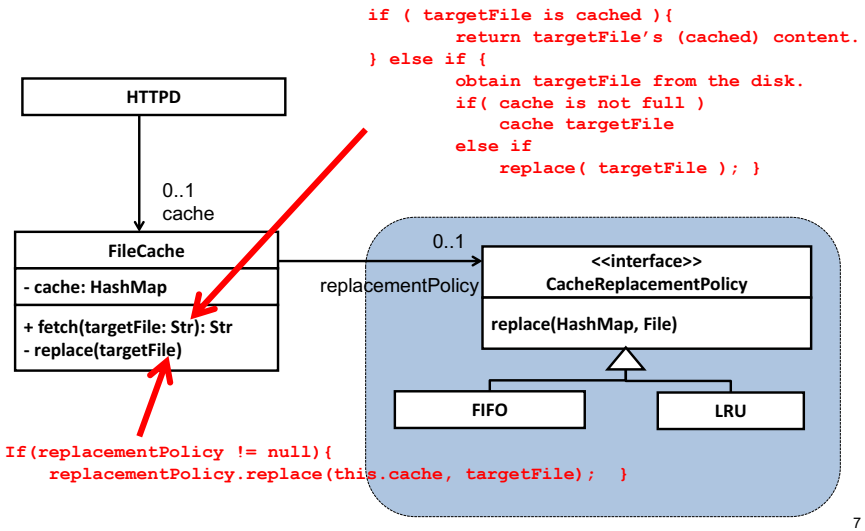

```
try{
    getAuthenticator().authenticate(...);
}catch(AuthenticationFailedException afe){
    makeErrorResponse(afe);
}
makeSuccessfulResponse(...);
```

Null Object as *Strategy*

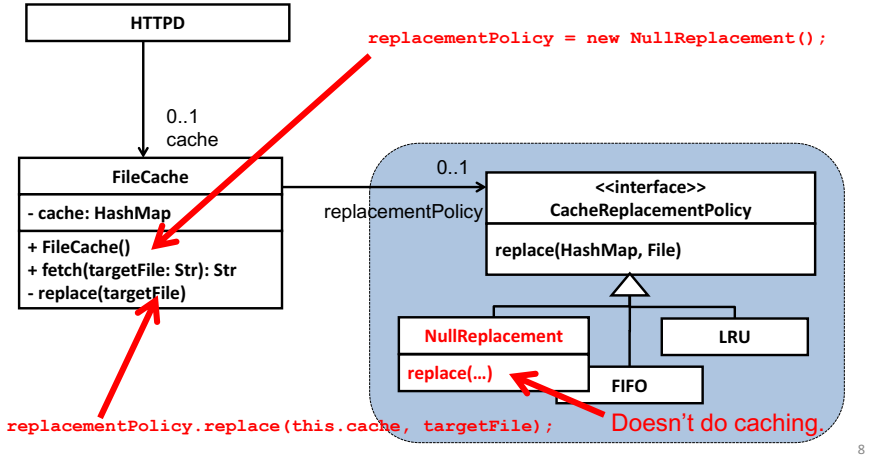
- Null object
  - A variant/application of *Strategy* that focuses on "doing nothing" by default.

Another Example: File Caching





File Caching with *Null Object*



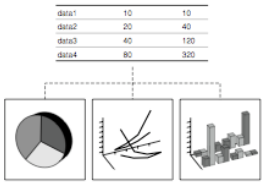
HW 7

- Implement a file caching mechanism with NullReplacement and FIFO.
  - Implement FileCache as a singleton class.
  - Have FileCache use NullReplacement by default.
  - Allow FileCache to dynamically change its replacement policy to FIFO.

- DUE: Nov 28 (Tue) midnight

Observer Design Pattern

Observer Design Pattern



Observable (publisher, event source)

registration/ subscription for an event (state change)

Event notification/ publication

Observers (subscribers, event listeners)

- Separate data processing from data management.
  - Data management: Observable
  - Data processing: Observers
    - e.g., Data analysis (e.g. feature extraction), graphical visualization, etc.

Observer Design Pattern

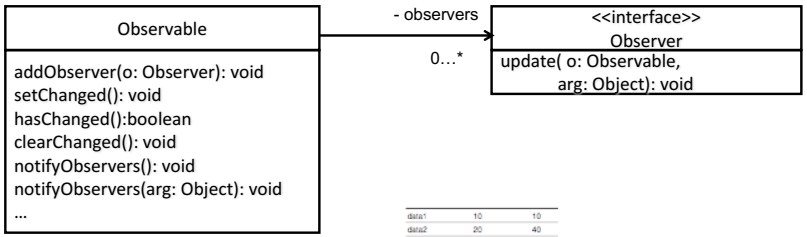
- Intent
  - Event notification
    - Define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified automatically

- a.k.a
  - Publish-Subscribe (pub/sub)
  - Event source - event listener

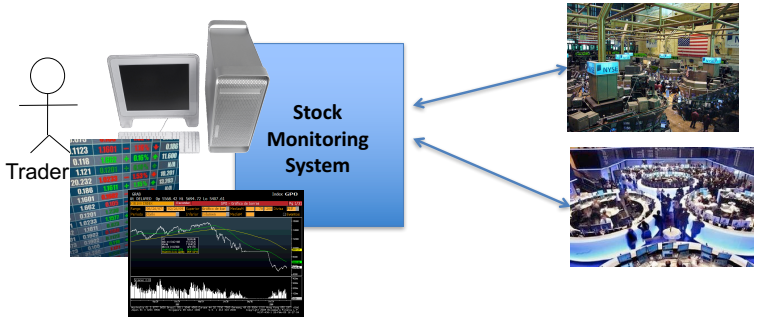
- Two key participants (classes/interfaces)
  - Observable (model, publisher or subject)
    - Propagates an event to its dependents (observers) when its state changes.
  - Observer (view and subscriber)
    - Receives events from an observable object.

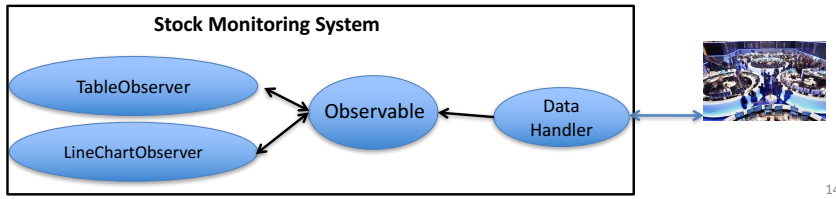
Class Structure

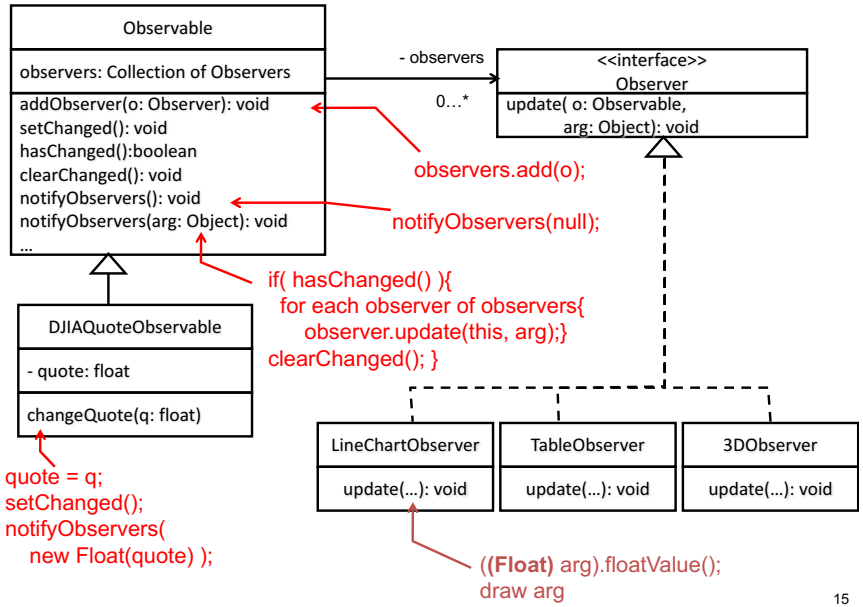
- java.util.Observable
- java.util.Observer

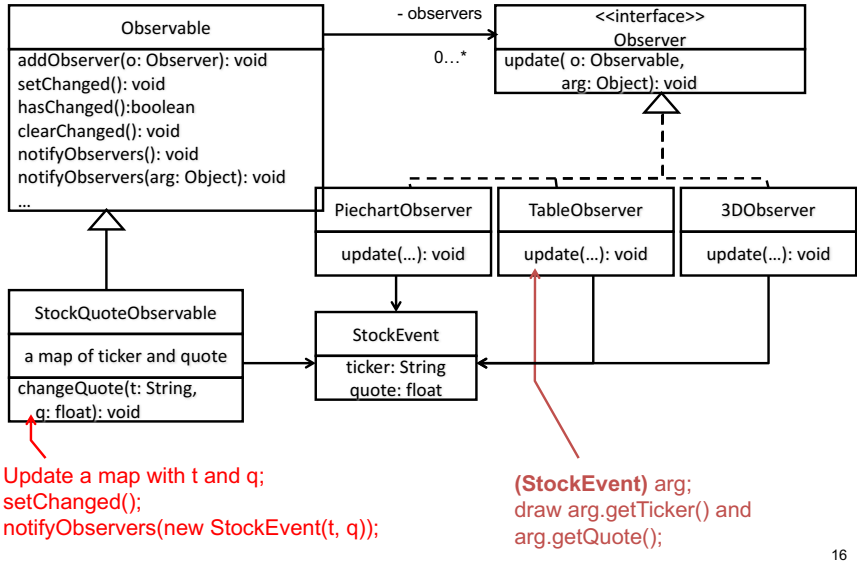


Example: Equity (Stock) Monitoring

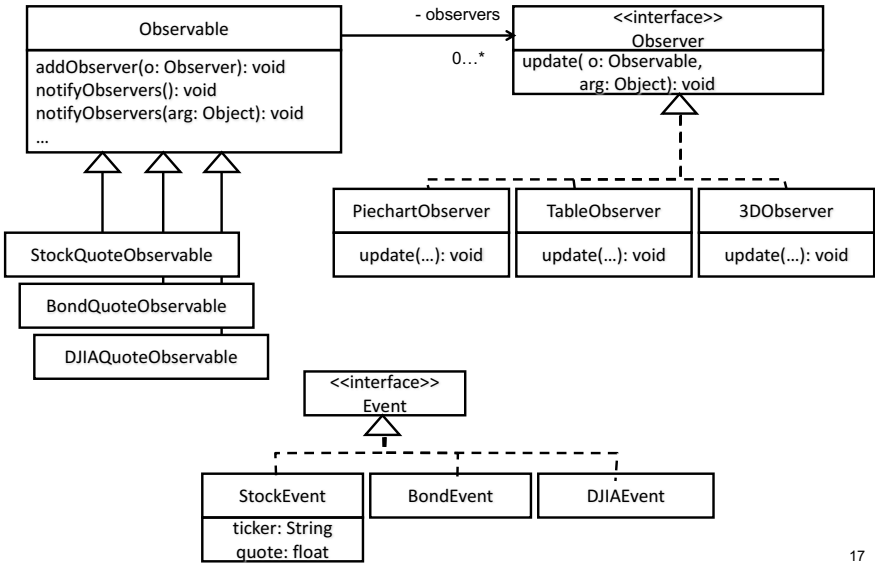


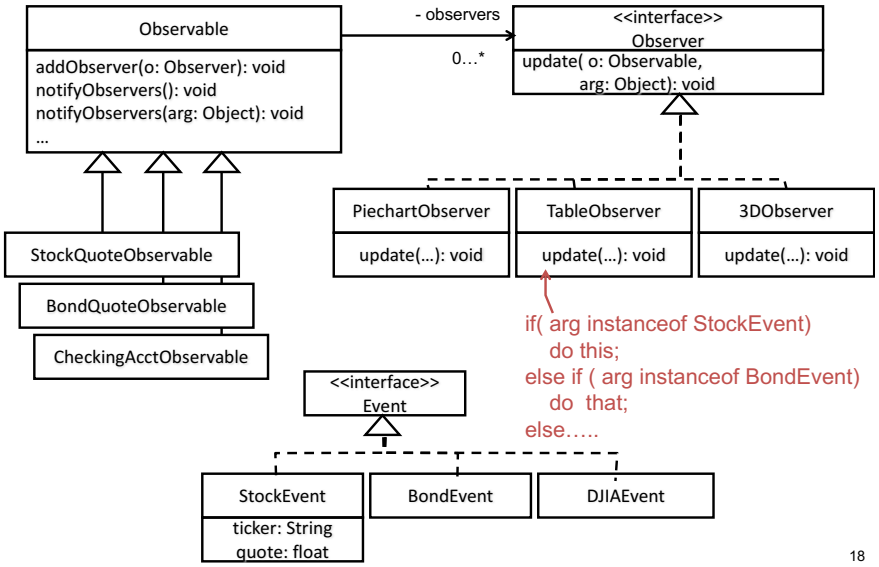






What if…

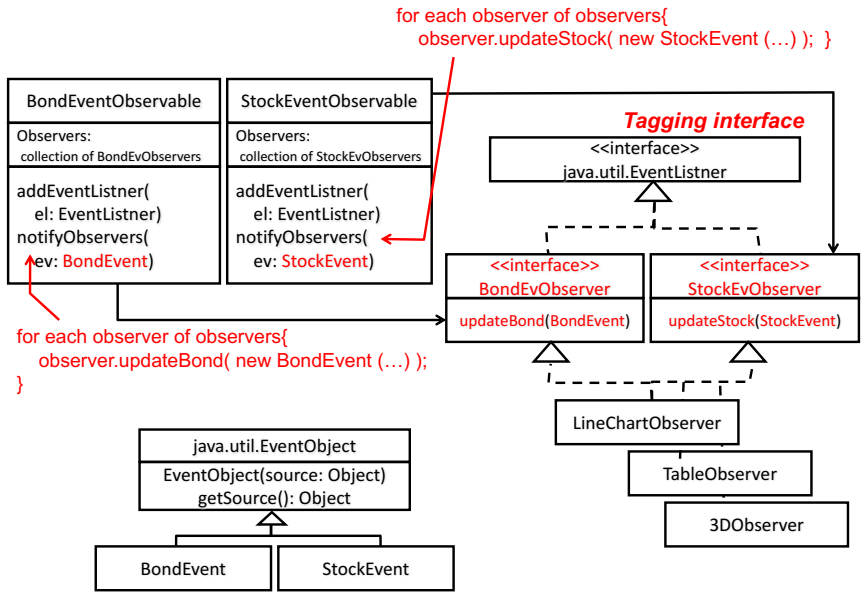




Multicast Design Pattern

Multicast Design Pattern

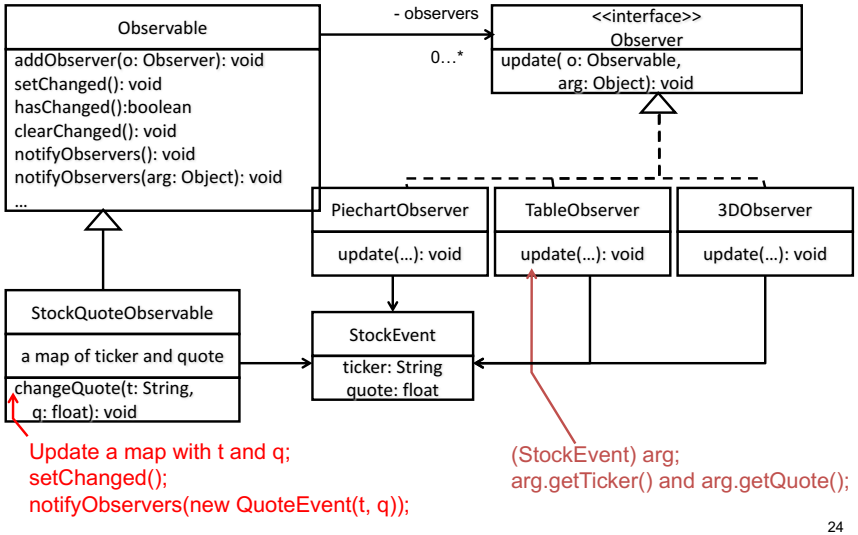
- Intent
  - Basically same as the intent of *Observer*
  - Focus on “many-to-many” event notification
  - Avoid conditional statements in observers.
- a.k.a.
  - “Typed” observer or “type-safe” observer



Which way to go?: *Observer* or *Multicast*??

- If you do many-to-many event notification, use *Multicast*.
  - If you use many kinds of observables, use *Multicast*.
    - to avoid writing a long sequence of conditional statements in observers
  - If you expect to add extra observables in the future, use *Multicast*.
    - to avoid maintaining/updating conditional statements in observers.
- Otherwise, use *Observer*.
  - Observer's design is a little bit simpler than Multicast.
  - But, Multicast is all right too.

HW 8: Implement this and its *Multicast* version



- You can reuse java.util.Observable and java.util.Observer.
- [OPTIONAL] Avoid using java.util.Observable and java.util.Observer and implement your own.
  - Use generics to define user-defined Observer and its update() so that you don't have to do downcasting in update().
- For a testing purpose, have an observable object randomly change its quote value periodically and notify quote changes to its observers.

- DUE: Dec 5 (Tue) midnight

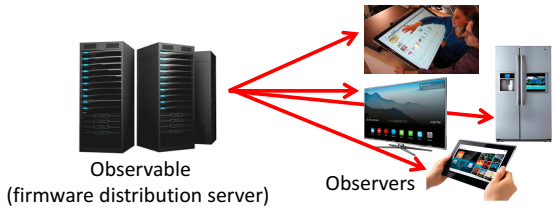
Half-Push/Half-Pull Design Pattern

"Push" and "Pull" in Event Notification

- Push
  - Observer and Multicast
  - Publish-subscribe (pub/sub)
  - Pros:
    - Low workload/traffic from observers to an observable
  - Cons:
    - An observable needs to perform error handling for unavailable (e.g., sleeping, turned-off or dead) observers.
      - An observable assumes that all observers are always-on by default.
      - Need to keep track which observers have received events.
- Pull
  - a.k.a. Polling
  - Observers (periodically) contact an observable to collect data/events.
  - Pros:
    - No error handling in an observable regarding unavailable observers.
  - Cons:
    - Potentially huge incoming workload/traffic on an observable.

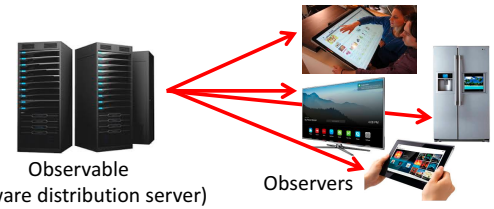
Half-Push/Half-Pull

- "Half-Push/Half-Polling," by Y. Son et al. PLoP '09, 2009.
  - http://www.hillside.net/plop/2009/papers/Security/Half-push-Half-polling.pdf
- Hybridization of push and pull event notification
- Example scenario
  - Firmware update on home appliances and consumer electronics.



- Pull
  - Each observer contacts an observable when it boots up.
  - If an event (e.g., a firmware update) is available, the observer downloads it from the observable or consults with the user.
  - Pros: No error handling necessary in the observable.
  - Cons: Huge incoming traffic on the observable.

- Push
  - Each observer registers itself to the observable.
  - Whenever an event is available, the observable pushes it to registered observers.
  - Pros: Limited incoming traffic on the observable
  - Cons: Need error handling in the observable. Need to keep track which observers have not responded and which observers have installed which versions of updates.



- Half-push/half-pull
  - Each observer registers itself to an observable.
  - Whenever an update (event) is available, the observable schedules when it provides the update to which observers. The observable *pushes* an "update schedule" to each observer.
    - The observable ignores unavailable observables. No error handling is performed.
  - According to a given schedule, each observer *pulls* (i.e., downloads) an update from the observable.
    - Update schedules need to be prepared carefully so that too many observers do not overwhelm the observable.
  - When an observer boots up, it requests an update schedule.
  - Pros: Modest incoming traffic on the observable. No error handling is necessary on the observable.