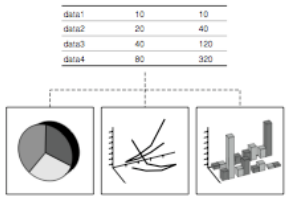
Observer Design Pattern

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.

An Example in Data Processing



Observable (publisher, event source)

Registration (subscription) for an event (state change)

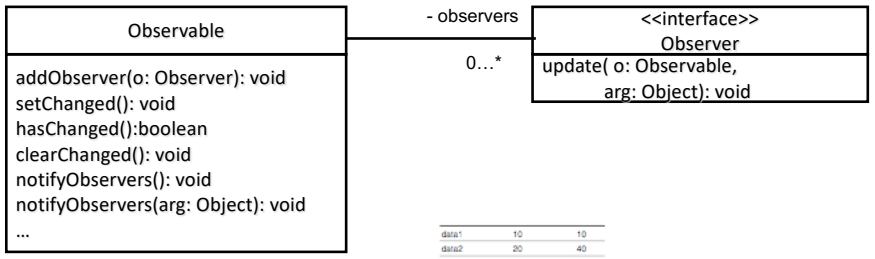
Event notification (publication)

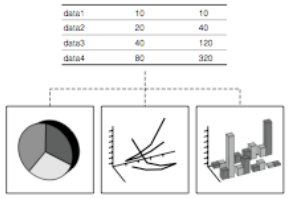
Observers (subscribers, event listeners)

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

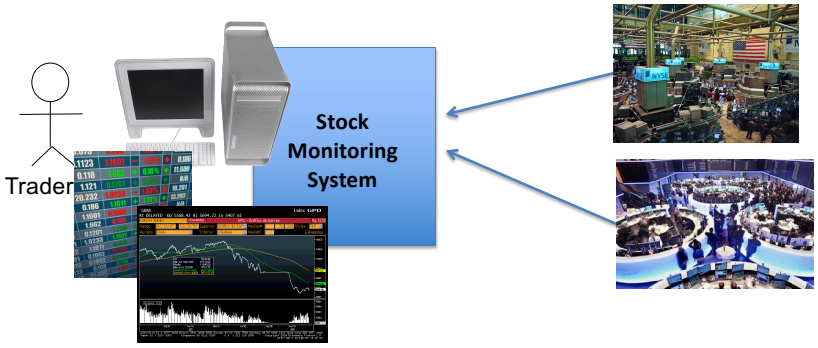
Class Structure

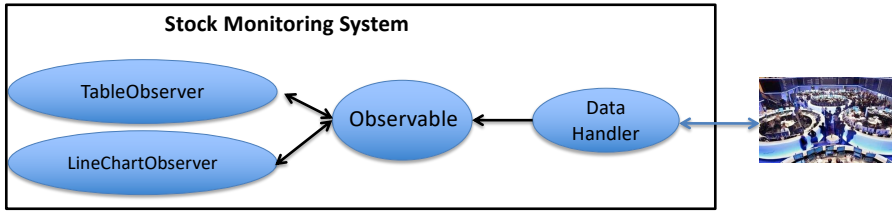
- `java.util.Observable`
- `java.util.Observer`

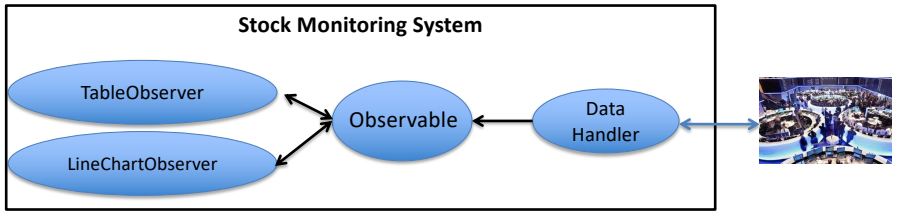




Example: Equity (Stock) Monitoring

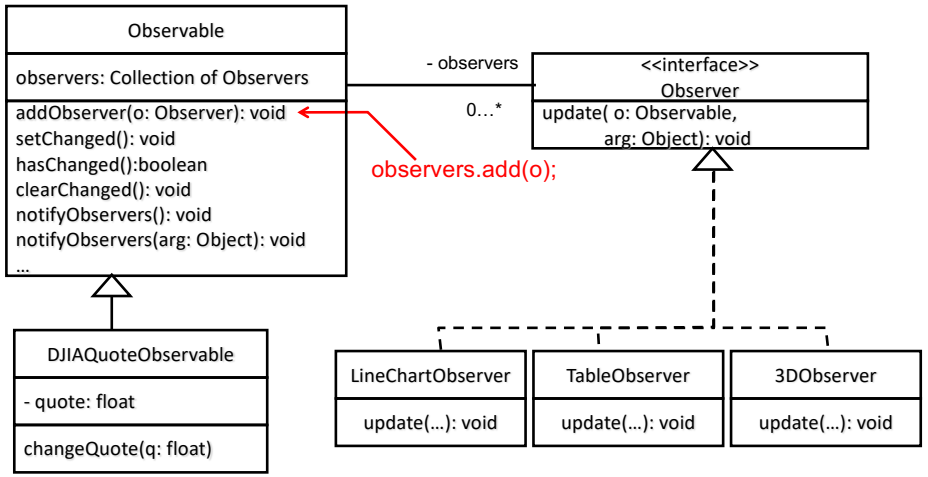






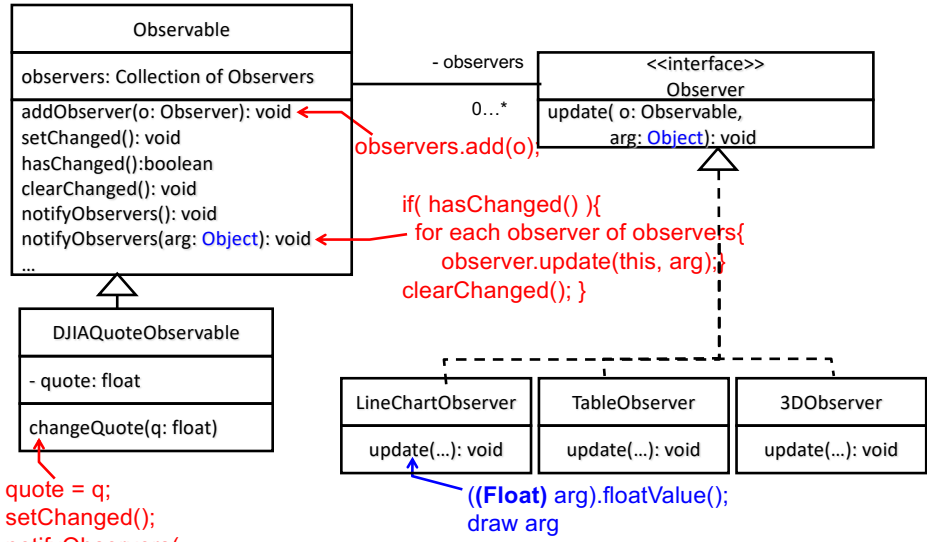
- Data handler: communication and data acquisition
  - Periodically fetches stock data from a stock exchange
  - Gives the data to an Observable
- Observable: event notification
  - Notifies the data to Observers
- Observer: data visualization
  - Displays the data

Registration

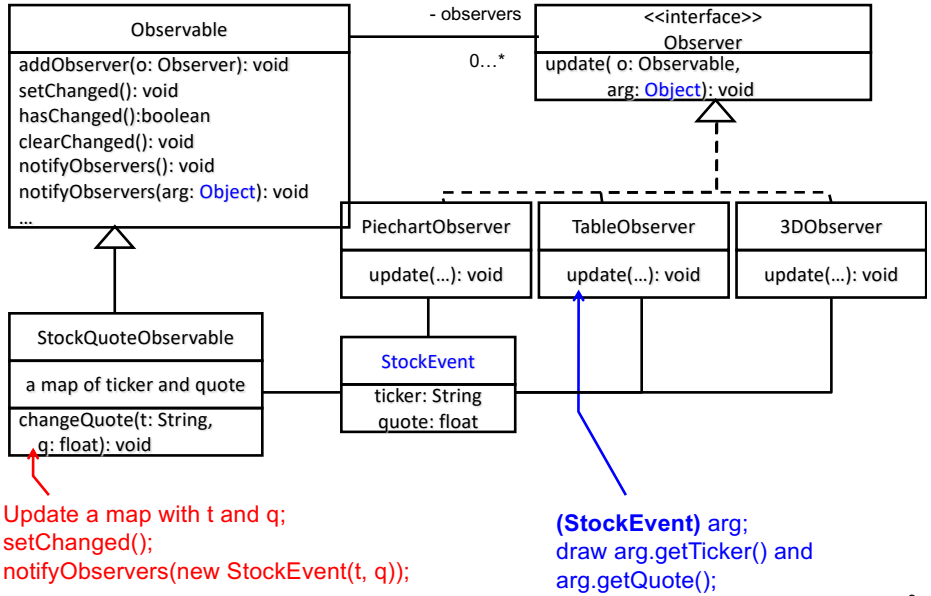


Each Observer calls addObserver() on DJIAQuoteObservable.

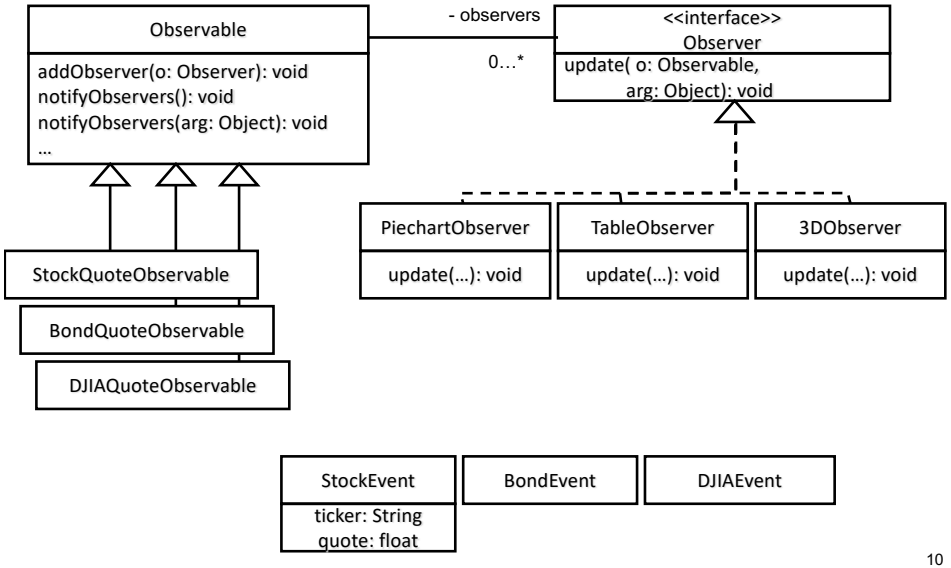
One-to-Many Event Notification

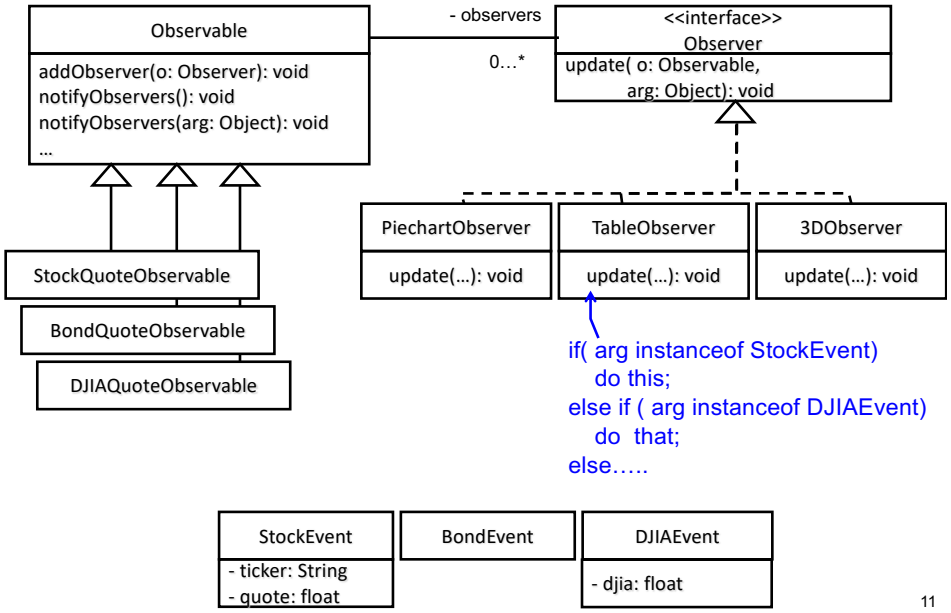


DJIAQuoteObservable calls notifyObservers()



Many-to-Many Event Notification

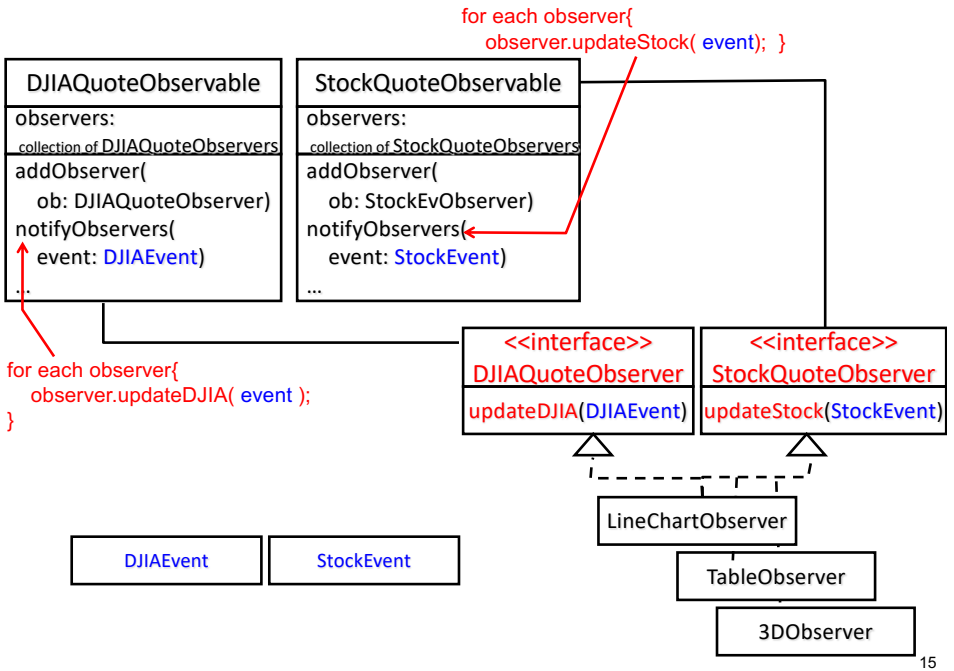




Multicast Design Pattern

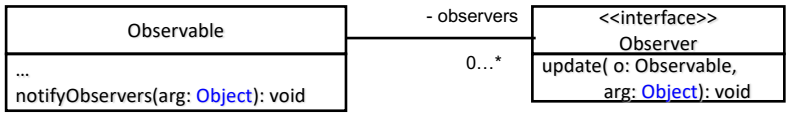
Multicast Design Pattern

- 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

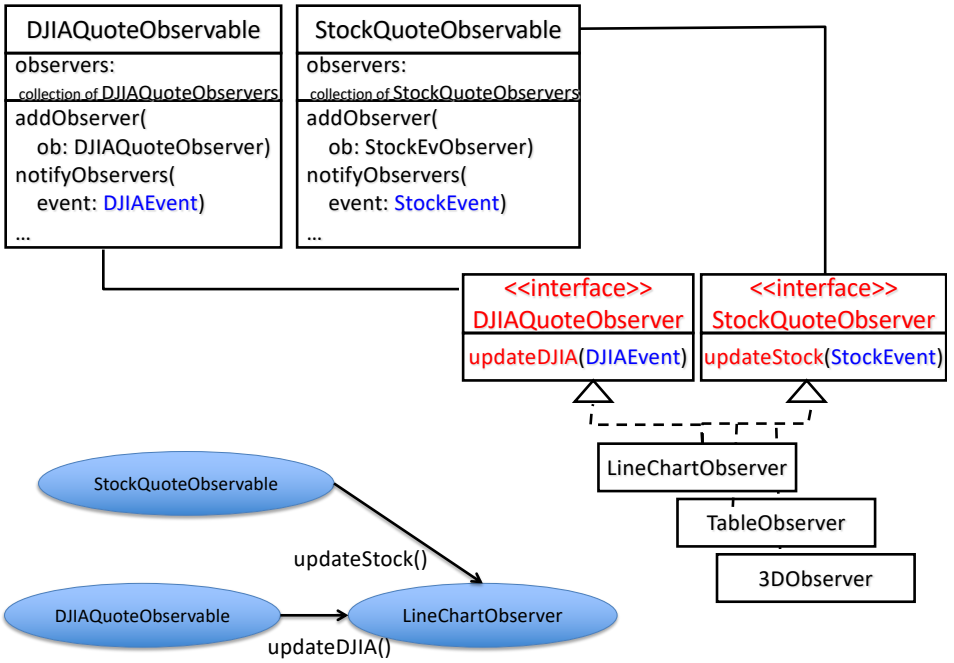


Points

- No longer uses `java.util.Observable` and `java.util.Observer`, which rely on `Object`.



- Define a pair of observable and observer for each event.
  - e.g., `DJIAQuoteObservable` and `DJIAObserver` for `DJIAEvent`
- Each observable/observer uses a specific event type.
  - e.g., `DJIAEvent`
- Each observer class can implement more than one observer interfaces,
  - so it can receive multiple types of events.



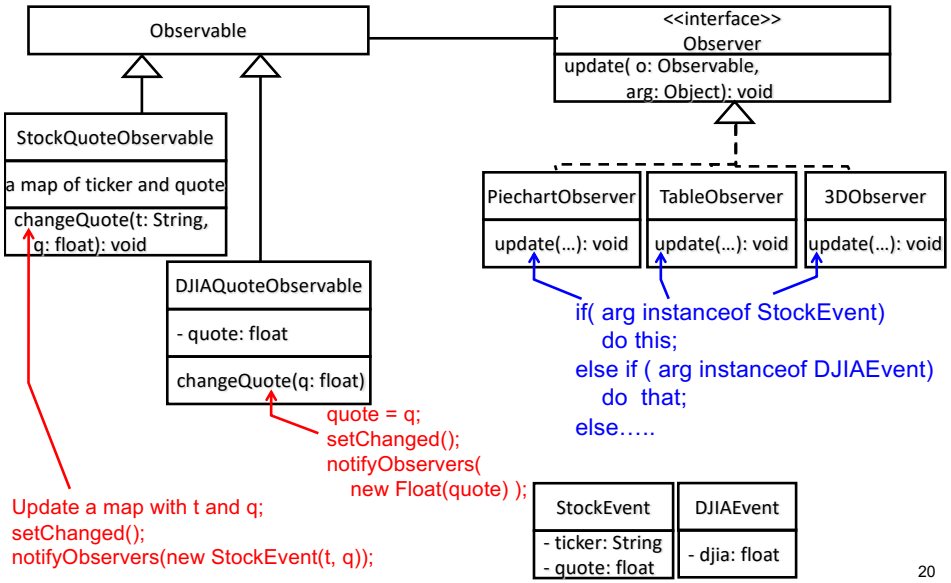
Pros and Cons

- Pros
  - Downcasting and conditionals are no longer required in observers.
- Cons
  - Need to define a pair of Observable (class) and Observer (interface) for each event.
  - Need to craft all classes/interfaces from scratch.
    - No reusable code (e.g. `java.util.Observable` and `java.util.Observer`) is available.

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 conditionals in observers
  - If you expect to add extra observables in the future, use *Multicast*.
    - to avoid maintaining/updating conditionals in observers.
- Otherwise, use *Observer*.
  - Observer's design is a little bit simpler than *Multicast*.
  - But, *Multicast* is all right too.

HW 11: Implement this *Observer* example AND its *Multicast* version

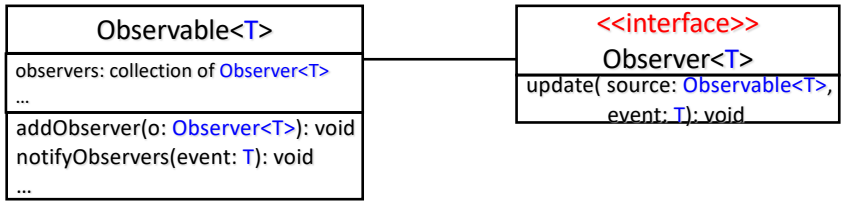


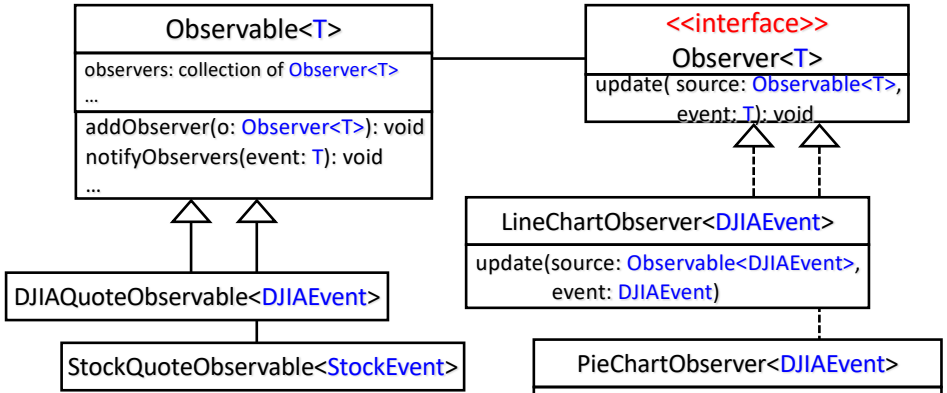
- *Observer*
  - You can reuse java.util.Observable and java.util.Observer.
  - [OPTIONAL] Avoid using java.util.Observable and java.util.Observer, and implement your own.
- *Multicast*
  - DO NOT use java.util.Observable and java.util.Observer
  - Make sure that observer objects avoid conditionals.
- A test case(s)
  - Have an Observable object randomly change its quote value periodically and notify the quote change to its observers.

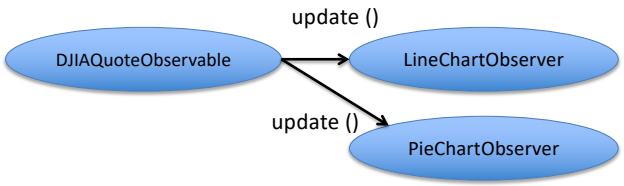
- Due: Dec 5 (Thu) midnight

HW 12 [OPTIONAL]

- Implement an improved version of `java.util.Observable` and `java.util.Observer`
  - Use generics to parameterize/specialize Observable and Observer with an event type (T), so that…
    - No downcasting is necessary in update().
- Use the enhanced ones to implement the *Observer* design pattern

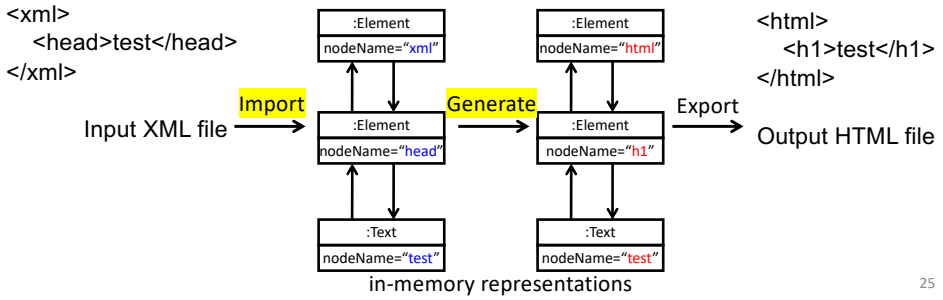




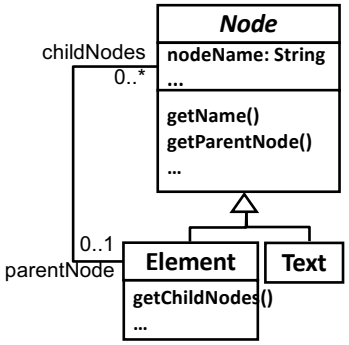


Quiz: Document Converter

- Suppose you are developing a conversion tool that
  - Imports/parses an (input) XML file and builds its in-memory representation with the Document Object Model (DOM) API
  - Traverses the in-memory representation to generate another in-memory representation for an (output) HTML file.



Simplified DOM API

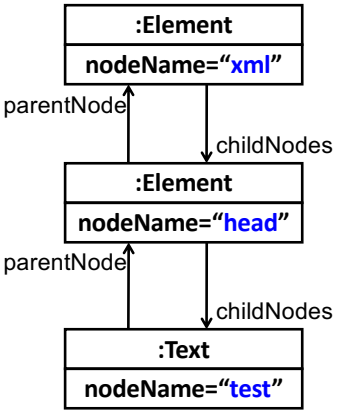


Element:

a textual block that starts with an opening tag (e.g. <xml>) and ends with its corresponding closing tag (e.g., </xml>).

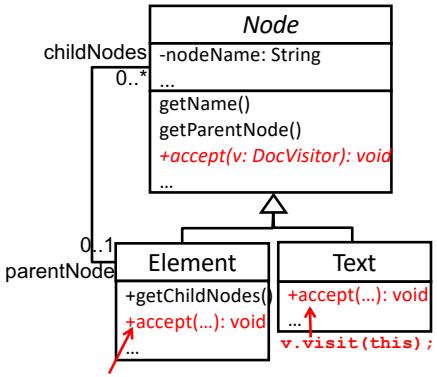
```
<xml>
   <head>test</head>
</xml>
```

Import

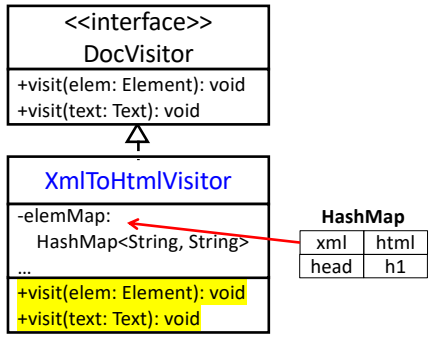


Input XML file

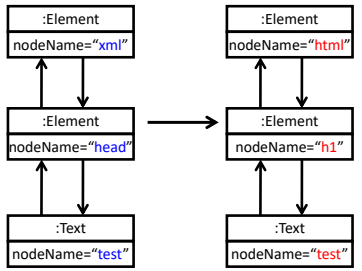
In-memory representation

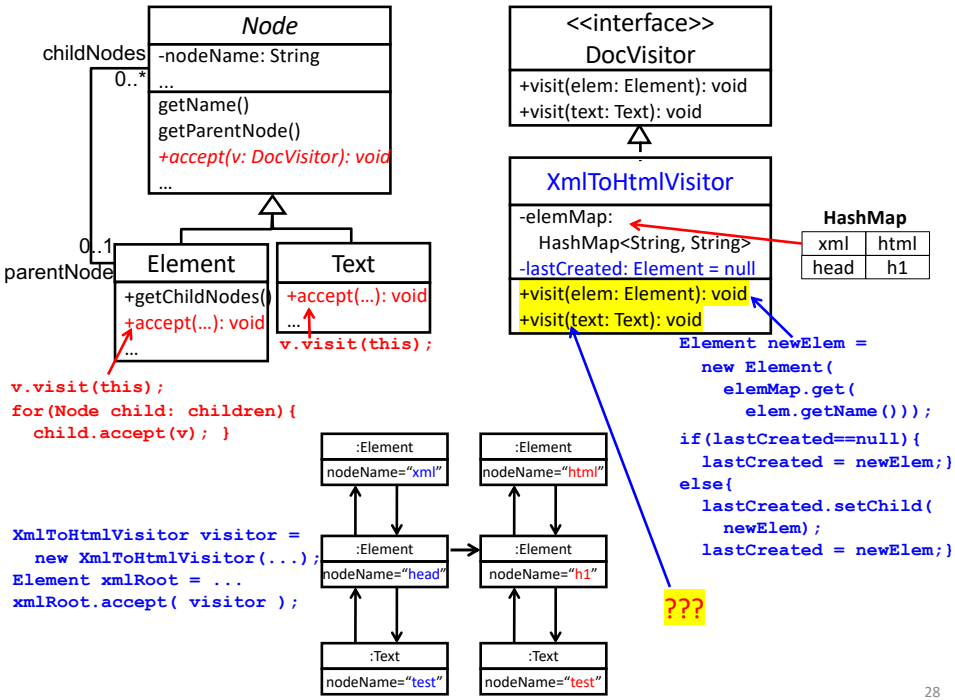



```
v.visit(this);
for(Node child: children){
  child.accept(v); }
```




```
XmlToHtmlVisitor visitor =
    new XmlToHtmlVisitor(...);
Element xmlRoot = ...
xmlRoot.accept( visitor );
```



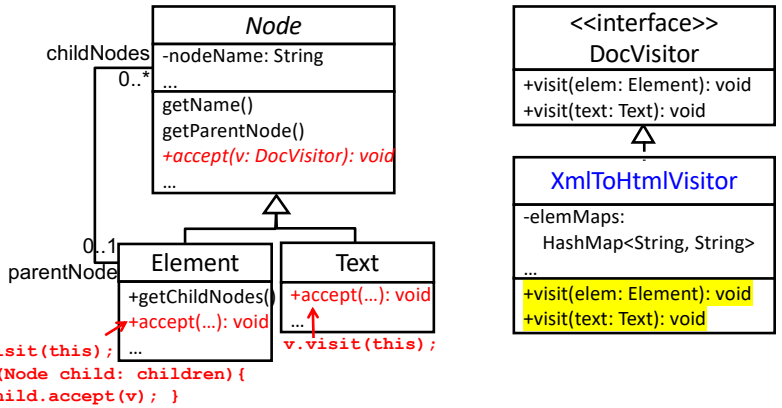



```
v.visit(this);
for(Node child: children){
  child.accept(v); }
```

```
XmlToHtmlVisitor visitor =
  new XmlToHtmlVisitor(...);
Element xmlRoot = ...
xmlRoot.accept( visitor );
```

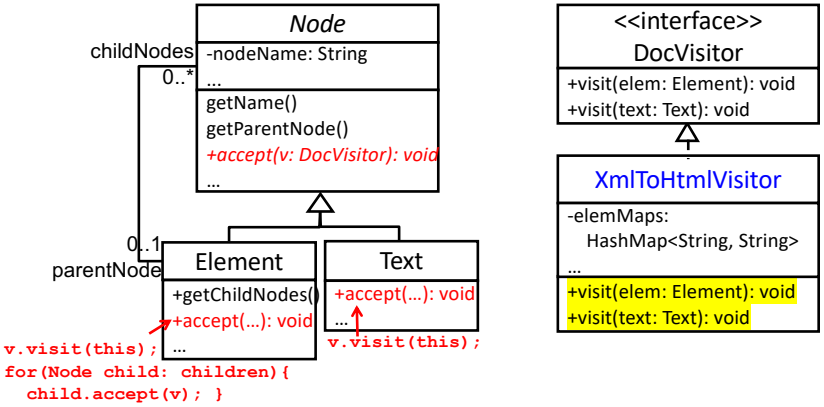
```
Element newElem =
  new Element(
    elemMap.get(
      elem.getName()));
if(lastCreated==null){
  lastCreated = newElem;}
else{
  lastCreated.setChild(
    newElem);
  lastCreated = newElem;}
```


Recap: *Visitor*



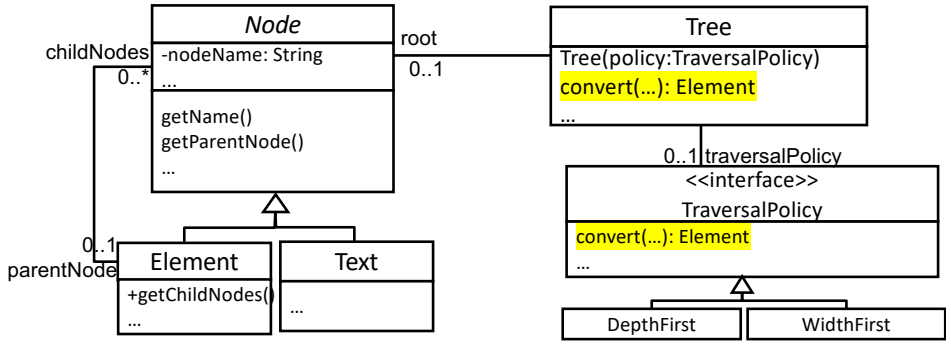

```
v.visit(this);
for(Node child: children){
  child.accept(v); }
```

- This design separates a set of foundation objects and the operations to be performed on those objects.
  - You can implement those operations (i.e. doc conversion operations) in a pluggable way (i.e. without changing foundation objects)



- However, this design does NOT allow you to implement tree traversal algorithms in a pluggable way.
  - A depth-first traversal algorithm is tightly-coupled with foundation objects (i.e. hard-coded in `Element.accept()`).

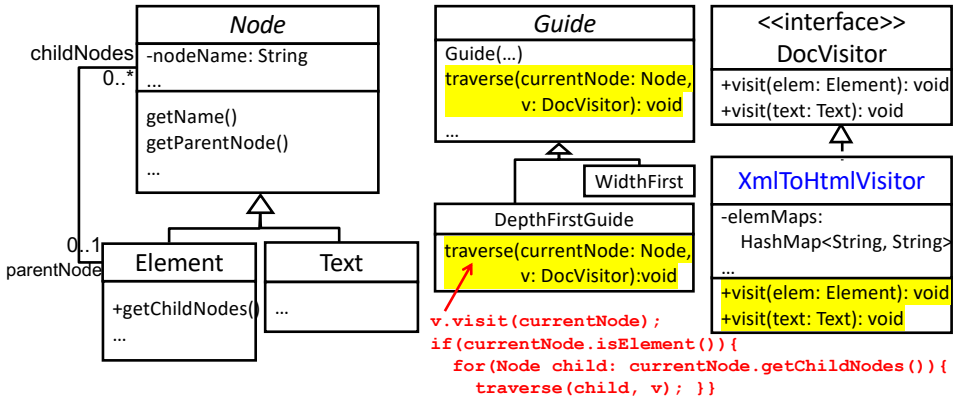
Recap: *Strategy*



- To separate traversal algorithms from foundation objects, consider to use *Strategy* (c.f. lec note on *Strategy*).
  - You can implement traversal algorithms in a pluggable way (i.e. without changing foundation objects).
- However, this design does NOT allow you to implement document conversion operations in a pluggable way.
  - A doc conversion operation are tightly-coupled with a traversal algorithm (i.e. hard-coded in `convert()`).

Towards a More Flexible Design

- Is it possible to separate
  - Foundation objects,
  - Traversal algorithms, and
  - Document conversion operations,
  - so that you can implement BOTH traversal algorithms and doc conversion operations in a pluggable manner?
- Use *Strategy* and *Visitor* at the same time.



- Each "guide" specializes in a specific traversal algorithm. It guides a visitor to visit nodes in a tree based on that algorithm.

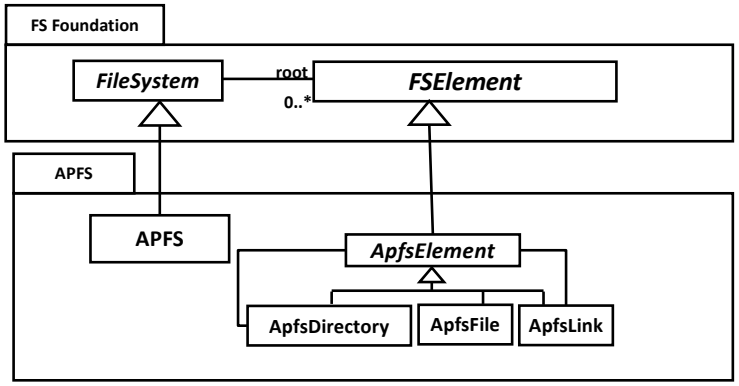
```
XmlToHtmlVisitor visitor = new XmlToHtmlVisitor(...);
Element xmlRoot = ... ;
Guide guide = new DepthFirstGuide(...);
guide.traverse( xmlRoot, visitor );
```

- If `traverse()` is defined as a static method…

```
XmlToHtmlVisitor visitor = new XmlToHtmlVisitor(...);
Element xmlRoot = ... ;
DepthFirstGuide.traverse( xmlRoot, visitor );
```

Recap: Apfs

- c.f. Previous HW

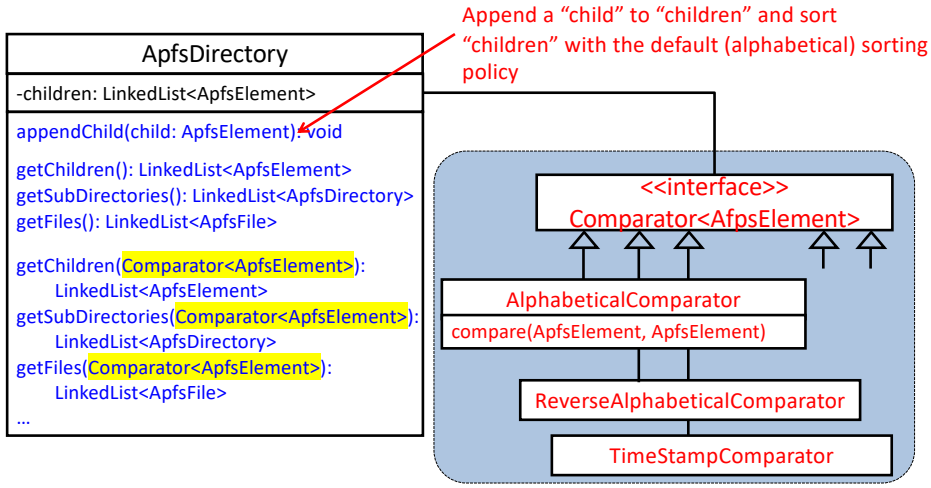


HW 13: Sorting FS Elements

- In prior HWs, FS elements were never sorted.
- Have APFS sort FS elements.
- Example soring policies
  - Alphabetical
  - Reverse alphabetical
  - Timestamp-based (e.g. “last-modified date”-based)
  - Element kind based (e.g. directories listed first followed by files and links)

- It is not a good idea to hardcode sorting logic in ApfsDirectory.
  - Whenever a new sorting policy is required, you need to modify ApfsDirectory.
- Better idea: Make ApfsDirectory loosely-coupled from sorting policies
  - Allow each FS user to select a sorting policy dynamically
  - Allow FS developers to add new sorting policies in a maintainable manner.
    - Have them add extra code (classes) rather than modify ApfsDirectory.
- Solution: Use *Strategy* (Comparator).

Soring FS Elements with Comparator



Strategy design pattern

- `getChildren()`
  `getSubDirectories()`
  `getFiles()`
  - Returns a `LinkedList` whose elements are sorted with the default (alphabetical) sorting policy.

- `getChildren(Comparator<ApfsElement>)`
  `getSubDirectories(Comparator<ApfsElement>)`
  `getFiles(Comparator<ApfsElement>)`
  - Re-sorts FS elements based on a custom (non-default) sorting policy, which is indicated by the method parameter, and returns re-sorted FS elements.
    - Use `Collections.sort(Comparator)`.
    - ApfsDirectory does not have to retain the re-sorted elements.
    - Implement at least one custom sorting policy

- Due: Dec 12 (Thu) midnight

Half-Push/Half-Pull Design Pattern

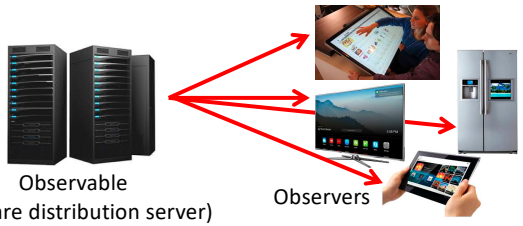
"Push" and "Pull" in Event Notification

- 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.

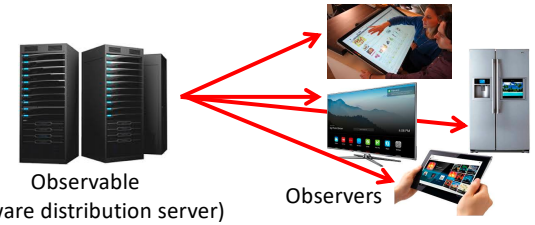
- Pull
  - a.k.a. Polling
  - Observers (periodically) contact an observable to collect data/events.
  - Pros:
    - No error handling in an observable regarding unavailable observers.
  - Cons:
    - Potentially huge incoming workload/traffic on an observable.

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.



- 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.



- 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.