COM S 309

OBSERVER PATTERN

Contents

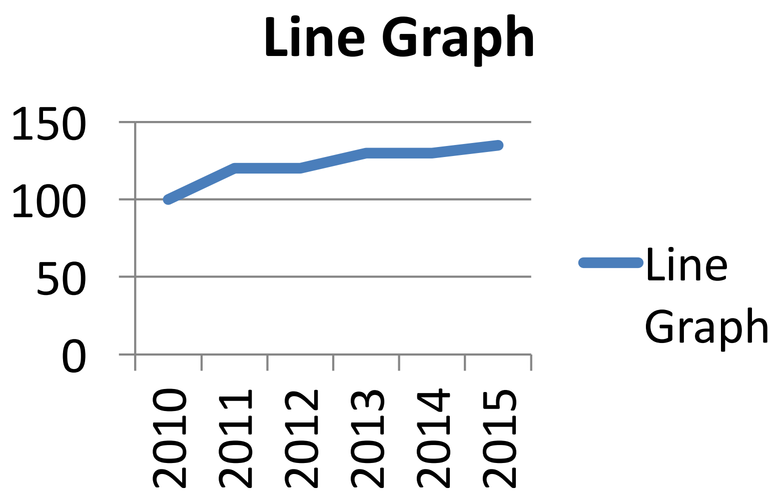
Problem

- Given two objects. One of them (observer) wants to know when something happens to the other object (the subject).
- Example: LineGraph (the observer) wants to know when table entries (the subject) are changed.

SUBJECT

Year	No Trees
2010	100
2011	120
2012	120
2013	130
2014	130
2015	136

OBSERVER



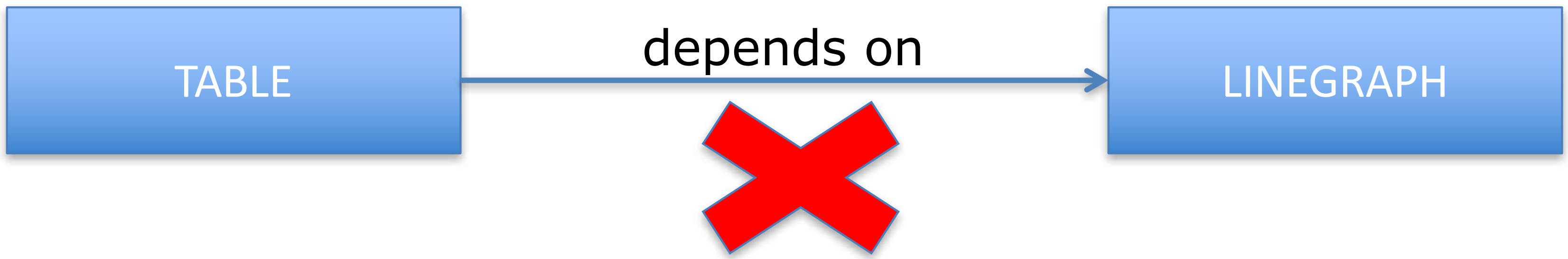
Problem continued

- One solution – polling! (why not a good idea?)

- Elements of a good solution:
  1. Subject code should be unaware of specific observers (and their specifics) – why?
  2. **Many observers** should be able to observe the same subject.
  3. An observer should be able to **observe multiple subjects.**

Example of BAD thing:
Subject code is aware of specific observer

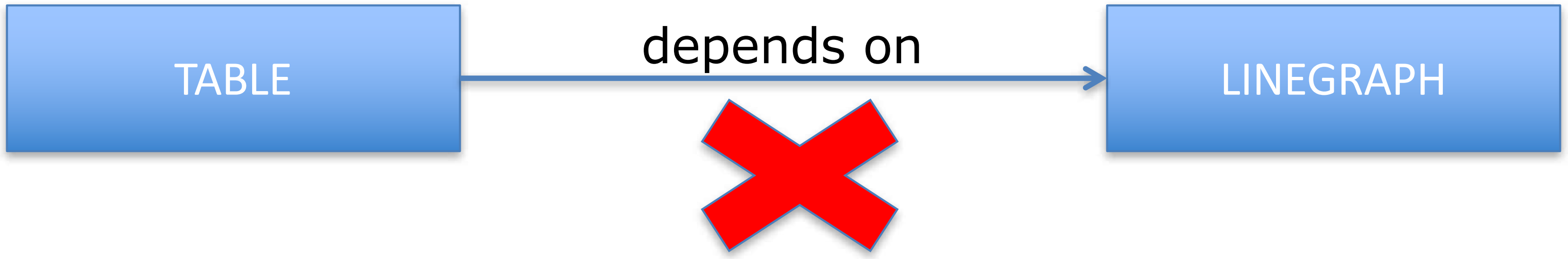
- In code for Table, there is a reference to LineGraph object so that LineGraph can be notified.

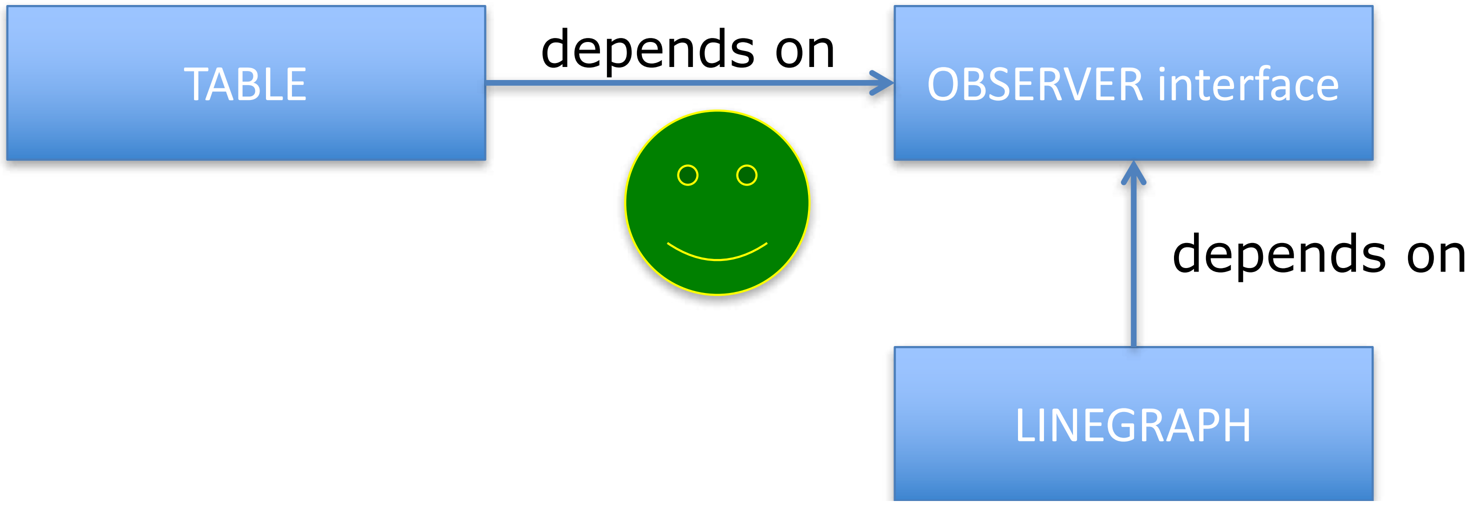


- Why is this a bad idea?
  - equivalent to hardcoding constant. Any change in name or method-name will have to be made in Table class too. NOT a modular design.
  - what if there are multiple observers? Then Table will depend on many classes directly. NOT a modular design.

Example of BAD thing:
Subject code is aware of specific observer

- In code for Table, there is a reference to LineGraph object so that LineGraph can be notified.

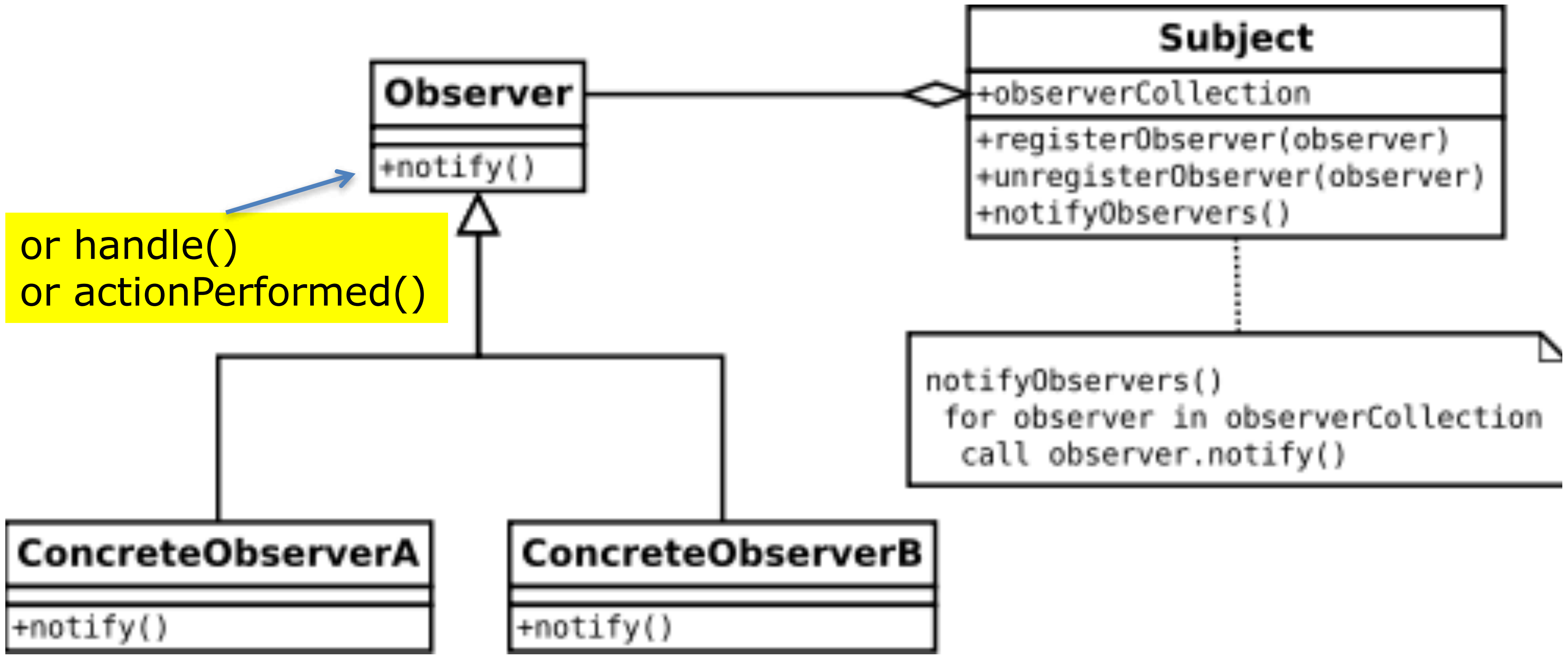




Solution Idea

- Subject has a list of observers (observer is an interface that specific observer objects will have to implement – that's why subject does not know about specific observers).
- When some change happens a fireEvent or notifyObservers method is called.
  - This method goes over the list of observers and calls their handle/notify methods one by one.
- observers can register/unregister themselves from a subject at any time.

Class Diagram



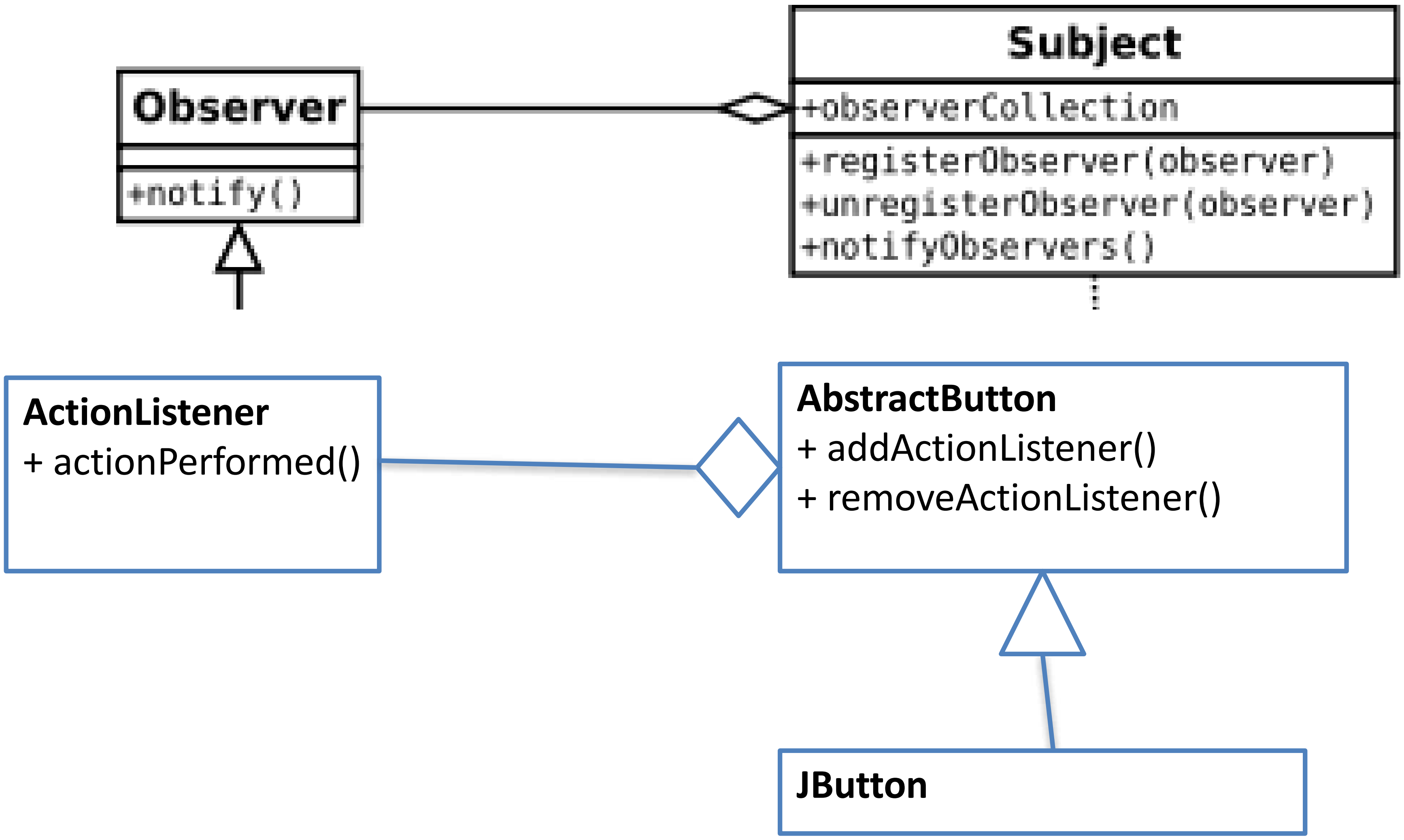
```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

CONCRETE example

- Jbutton is Subject
- addActionListener() is registerObserver()
- removeActionListener() is unregisterObserver()

- ActionListener Interface is **Observer**
- actionPerformed method is **notify()**

Class Diagram



BENEFITS

Original elements of a good solution (on slide 4):

1. Subject code should be unaware of specific observers (and their specifics) – why?
2. **Many observers** should be able to observe the same subject.
3. An observer should be able to **observe multiple subjects.**

1. when subject is changed (or event happens)… all observers are notified of change automatically – the MAIN requirement!
2. subject doesn't know about specific observers (they are very loosely coupled) – meets #1 requirement.
3. Also, observers code has no reference to any specific subject!
4. Many observers can observe the same subject - meets #2 requirement
5. An observer can observer multiple subjects for events – meets #3 requirement.
6. new observers can be added and removed at any time without ANY change in code for subject.

What you need to remember

- You can use this pattern in your own code as well!
  - Note that you can use **java.util.Observer** and **java.util.Observable** interface in your own code (i.e. non Java Swing code)
- Understand all the benefits of using this pattern.

SELF CHECK

Q1. **stockholder** wants to know whenever **stock** price goes below a specific number n

Show code snippets needed using java.util.Observer etc.

Q2. Enumerate benefits of Observer Pattern.