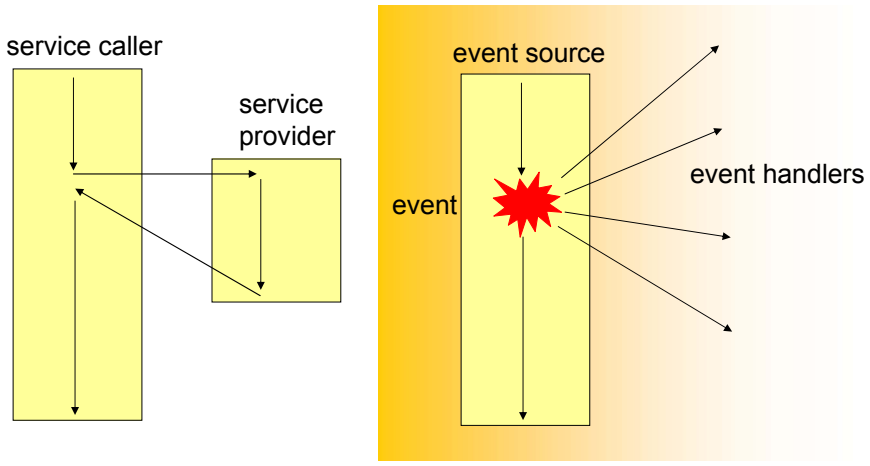
# Observer

GoF Behavioral Pattern
Responsibility pattern

## Background for Observer
## From services to events

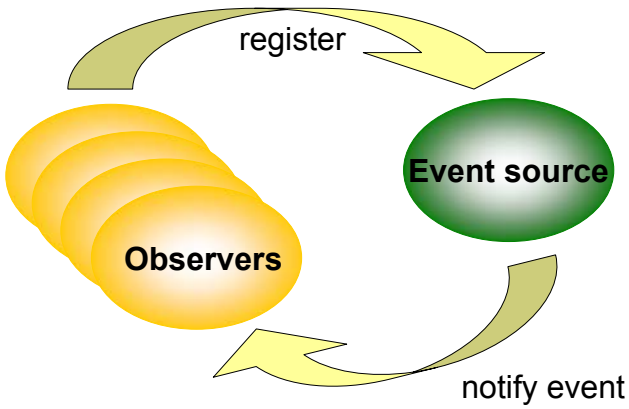



- service callers arrival in some place of its code can bee seen as a global event, related modules can react in various ways
- neither the event producer, nor handler need to know each other, get rid of direct dependency

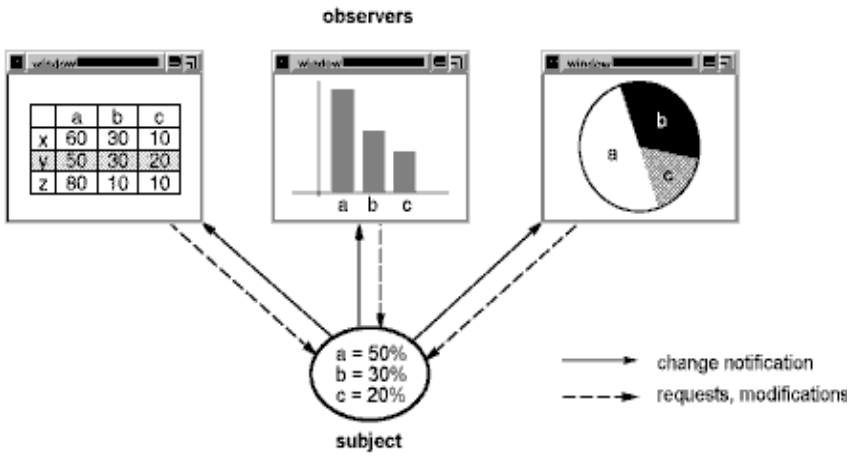
## Dimensions of event-based systems

- synchronous or asynchronous control flow?
- causing and handling an event in same (synchronous) or different processes?
- where are the observers registered, locally or globally?
- who determines the time of handling the event, source or observer?
- does an event carry additional information?

## Example: synchronous event-handling, source notified system with source-registered observers



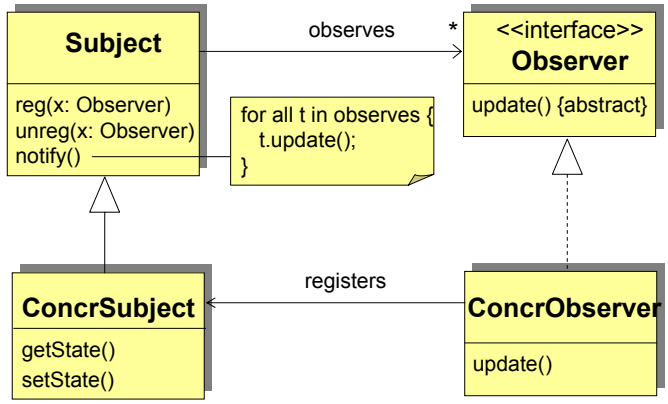
Example: synchronous event-handling, source notified system with source-registered observers



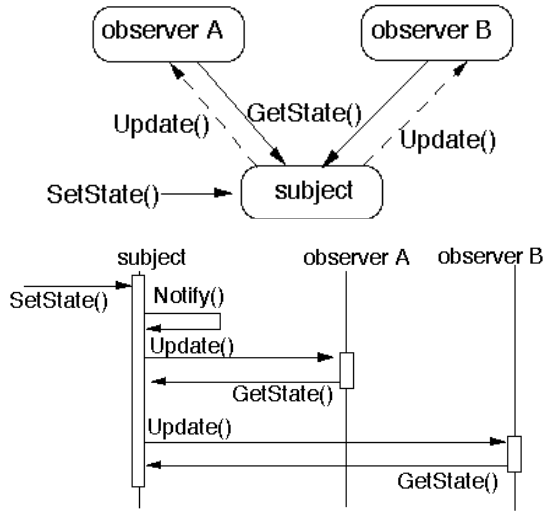

## Intent and problem

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Problem**
  - A large monolithic design does not scale well as new dependent objects are added.
  - The reusability of classes is lost in a dependent design.

# Observer design pattern




# Collaboration



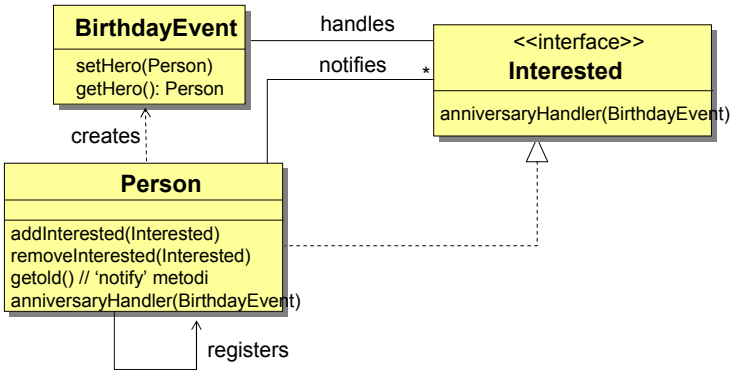
## Applicability

- Use the Observer pattern:
  - When an abstraction has two aspects, one dependent on the other. You need to encapsulate these aspects into separate objects for e.g. to achieve reusability.
  - When a change to one object requires changing others, and you don't how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about who these objects are.
  - The control model is of type synchronous event-handling, source notified system with source-registered observers
- Observer is a widely used pattern, once you understand it, you see uses for it everywhere
  - You can register observers with all kinds of objects rather than writing those objects to explicitly call you.
- It is easy to use adapter to make an object fit the Observer interface

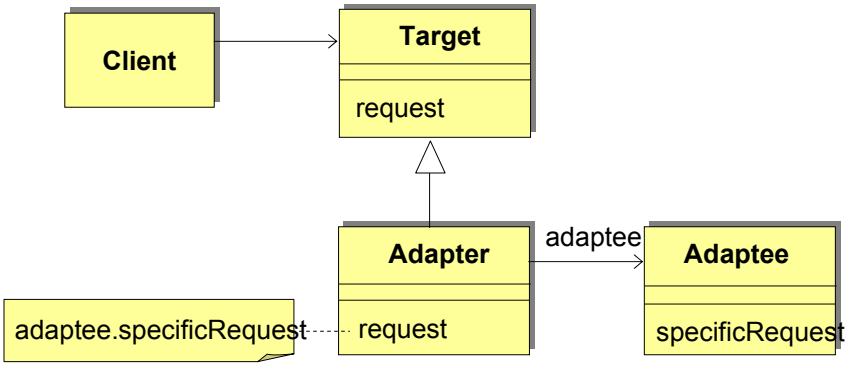
## Discussion

- How it works
  - Define an object that is the "keeper" of the data model and/or business logic (the Subject).
  - Delegate all "view" functionality to decoupled and distinct Observer objects.
  - Observers register themselves with the Subject as they are created.
  - Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.
  - This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.
- There are two primary of Observer pattern, *pull model* and *push model.*
  - the protocol described above specifies a pull interaction model. Instead of the Subject pushing what has changed to all Observers, each Observer is responsible for pulling its particular *window of interest* from the Subject.
  - The push model compromises reuse, while the pull model is less efficient.

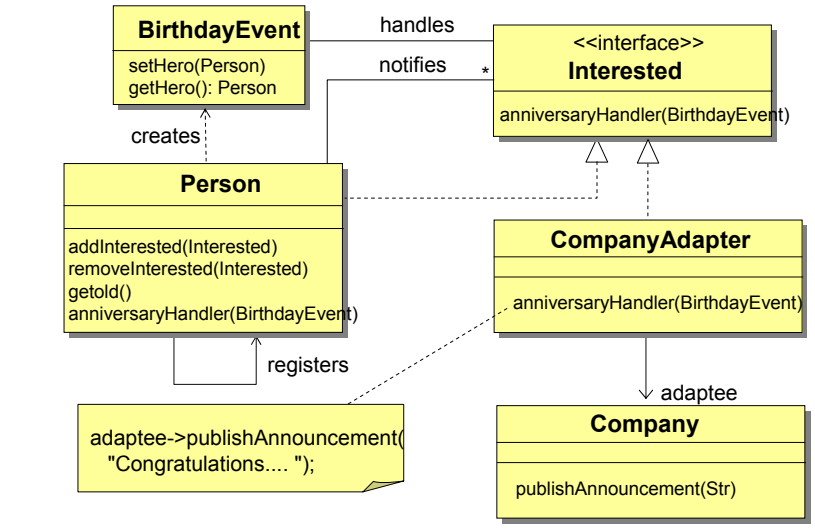
## Example: Observer – the same class acts both as an observer and subject




## Remeber this?
## Adapter design pattern



## Example: Observer and Adapter




company class is used by many clients, can not be changed e.g. by making it implement ”intrested”

## Observer and OO design principles

- The main driving principle behind observer is OCP – the design is open for extending by adding new observers so that you do not have to change the observed object (Subject). Thus the observed object stays closed.
- In the birthday example, a Person is substitutable for interface Interested. The abstract class Subject is combined with the ConcreteSubject in class Person. Thus it is clear that Person is substitutable for Subject, satisfying LSP.
- DIP?
  - Observer is an abstract class, ConcreteObserver depends on it, OK.
  - Subject has only concrete methods and still ConcreteSubject depends on it. This does not violate DIP because Subject is not meant to be ever instantiated. Thus, Subject is *logically* abstract.