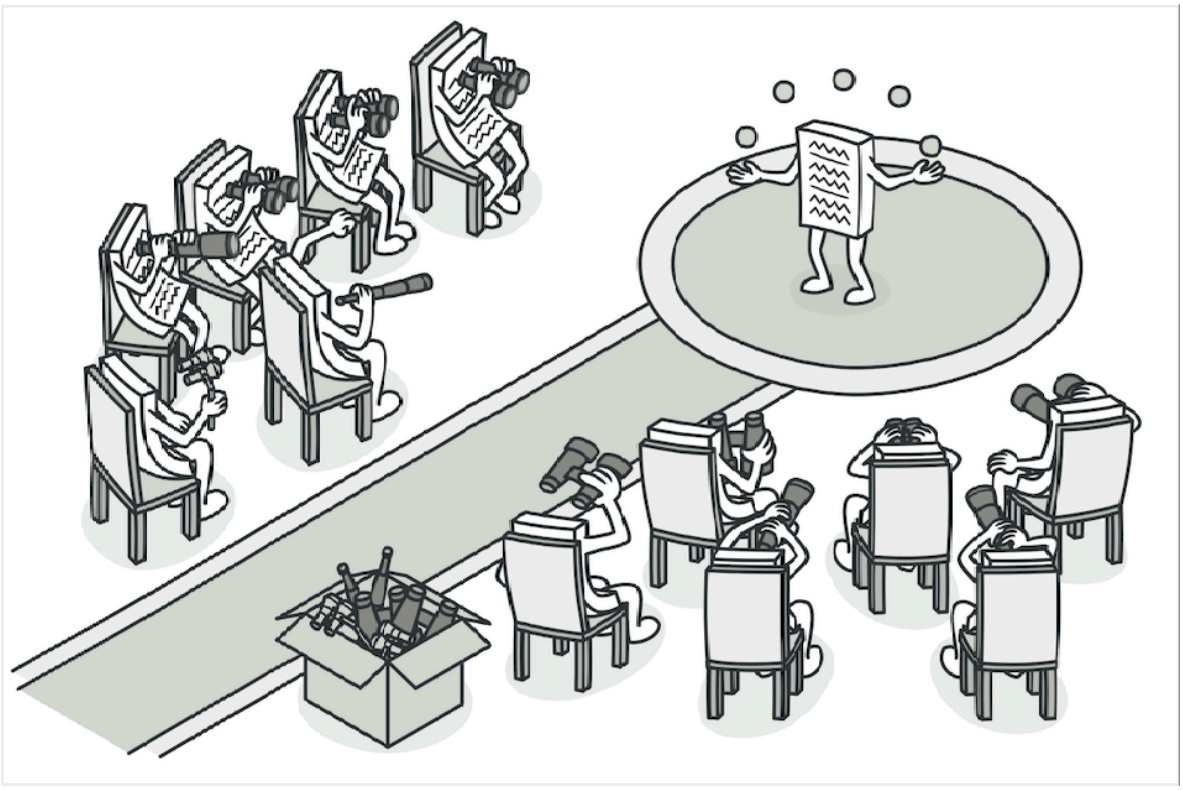
8. Observer Design Pattern

Definition

- Observer pattern lets us to define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing (listening to).



Problem

- We have an Amazon class, where we have a method called ***orderPlaced()*** which will be executed whenever an order is placed.
- The business requirement is, whenever an order is placed we have to,
  - Update the inventory using Inventory service
  - Generate the invoice using InvoiceService
  - Notify the seller about the order
  - Send the email notification to the customer
  - Send the app notification to the customer

Approaches

- ***Approach 1*** - Have all the different logics inside the ***orderPlaced()*** method.
  - Issues - Which leads to bulky code in the method & it violates SRP. And also if suppose the ***emailService*** is down & still we need to allow

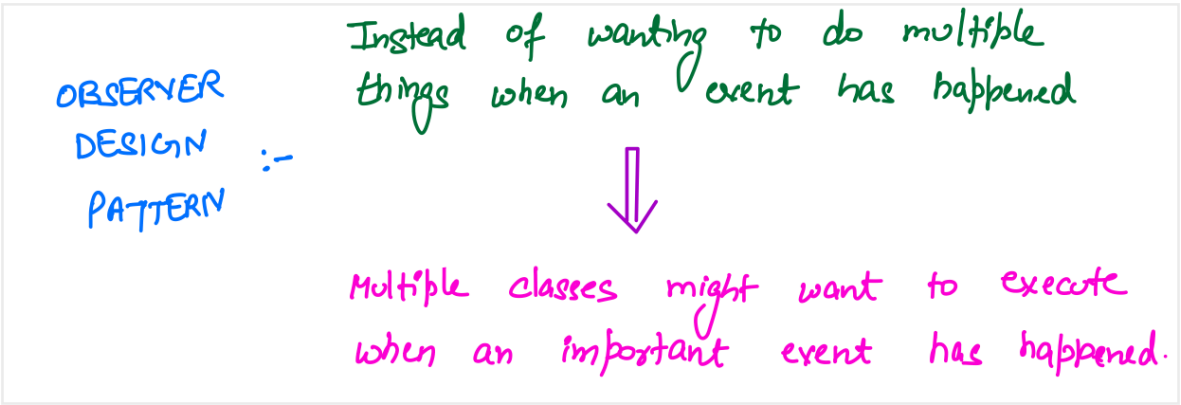
customers to place orders. In this case we have to manually add/remove that particular service from the code every time & recompile it and deploy. We can't able to add/remove any functionality at the run time.

- ***Approach 2*** - Have an ***OrderPlacedFacade*** helper class which will do all the business requirement when the order is placed, by this approach we can remove the bulky code inside the Amazon class's ***orderPlaced()*** method.
  - Issues - If suppose the ***emailService*** or any other service is down & still we need to allow customers to place orders. In this case we have to manually add/remove the particular service from the code every time & recompile it and deploy. We can't able to add/remove any functionality at the run time.

- ***Approach 3*** - Observer design pattern

Observer Design Pattern

- Instead of doing multiple things (generating the invoice, notify the seller, send email & app notification, updating the inventory) when an order has been placed at one place. Create separate classes for all different functionalities & execute their own logic when an order has been placed. Using Publisher - Subscriber.



Implementation

- Create an interface & make all the subscribers to implement it.
- Create methods in Publisher class (Amazon) that allows Subscribers to register/ deregister themselves.
- Maintain a list of subscribers in the publisher class for all the events separately, when an event has occurred loop through all the subscribers & notify them to do their logic (through the common method in the interface).
- Whenever creating an object of subscriber, make them to register themselves to the publisher.

By this approach, even when a service is down, we can have a API to register/deregister a particular service from the subscriber at the runtime without updating the code & recompile and deploy it.