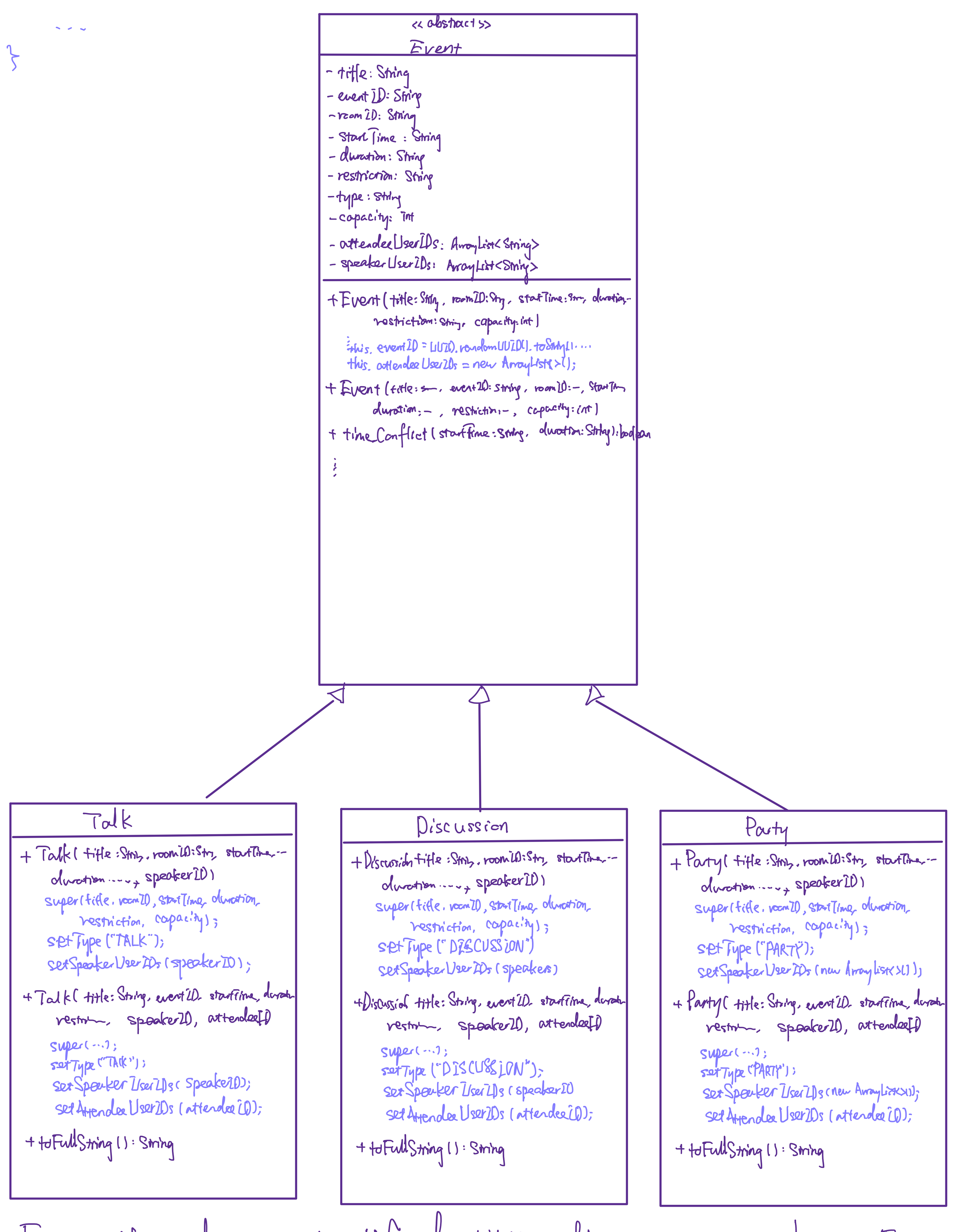
Factory-Method-Design-Pattern-1

Sunday, December 13, 2020 3:02 PM

In eventSystem/EventFactory.java, we implemented the Factory method i.e. the Simple Factory Design Pattern.

```
public class EventFactory {
    public Event createEvent(String type, String title, ....) {
        switch (type) {
            case "TALK":
                return new Talk(title, roomID, ...., speakerID);
            case "DISCUSSION":
                return new Discussion(title, ...., speakerID);
            case "PARTY":
                return new Party(title....);
        }
        return null;
    }
    ...
}
```



From the above simplified UML diagram, we have Event as our abstract superclass of the eventSystem, Event is an abstract class, therefore we cannot instantiate it directly. Instead, we would like to create instances of one of its subclass: Talk, Party, or Discussion.

This situation meets the condition to use the Factory method in that;

① . One class wants to interact with many possible related objects.

In eventManager, we have line 311:

```
public boolean createEvent(string type, .....){
    // validate data (checking time conflict)
    Event newEvent = eventFactory.createEvent(type, title, ...);
    events.add(newEvent);
    return true;
}
```

That is, in eventManager, we want to store all the events in its data field events, which is of type: List<Event>.
In other words, no matter what type of event we create, be it Talk, Discussion, or Party, it can be stored in this events List, as all the events that are in our system currently.

This is the driver code of the eventFactory, or the client code that uses the eventFactory to create an Event for us.
In this way, we encapsulate the details of object creation in its own factory class. Once this factory is ready, the createEvent() method just becomes a client of that object — Anytime it needs an event, it asks the eventFactory to make one.

②. We want to obscure the creation process for these related objects.

Notice that the type of newEvent is Event, which is the superclass type of the event we would like to create.
That is to say, in eventManager.java, we don't care much about whether the event is a Talk or a Party, as long as it is an event.
Also notice that the return type of the eventFactory.java's createEvent() method is also Event. So we don't know which subtype of Event we've just created, but we know it's one of those 3 types: Talk, Discussion, or Party.

The instruction for which one exactly we're creating came from the user, the type parameter passed in will decide the exact type of Event we're actually creating.

③ At a later date, we might want to change the types of the objects we are creating.

Our design is open for extension, if sometime later, we decide to add another type of Event, say "GroupMeeting", which is a subclass of Event, then the code in eventManager can stay the same without having to be modified, and we can isolate the change to only happen in the eventFactory, where we would only need to add very few code for the new type of event, like the following code in green.

```
public class EventFactory {
    public Event createEvent(String type, String title, ....) {
        switch (type) {
            case "TALK":
                return new Talk(title, roomID, ...., speakerID);
            case "DISCUSSION":
                return new Discussion(title, ...., speakerID);
            case "PARTY":
                return new Party(title....);

            case "GROUPMEETING":
                return new GroupMeeting(title, ....);
        }
        return null;
    }
    ...
}
```

By implementing the Factory method, not only is our project easy to be extended, we also follow the SOLID principle.

Since we put all the constructor calls (new Talk(...), or new Discussion(...), or new Party(...)) and the decision about which constructor to call into a separate class — EventFactory, and use that class to create instances of all childclasses of Event, we have followed:

①. Single Responsibility Principle:
The factory class has the single responsibility of creating instances of Events, i.e. this class has only one reason to change, or at most one source of potential change.

②. Open-Closed Principle:
As shown above, it's really easy to add new Events! In factory, you just add an extra couple of lines everytime you create a new type of Event. The return type of createEvent() method is still Event, and the rest of the code still works.

③. Liskov Substitution Principle:
When we're creating a new type of Event, we need to make sure the subclass of Event we create can replace Event. Which means that an instance of the subclass of Event can replace an instance of Event class without breaking the program.

Similarly, we use the Factory Method Design Pattern to help us create all types of users: