JAVA ACADEMY - XIDERAL MONTERREY, N.L

EXERCISE - WEEK 3 OBSERVER PATTERN

Made by:
Luis Miguel Sánchez Flores

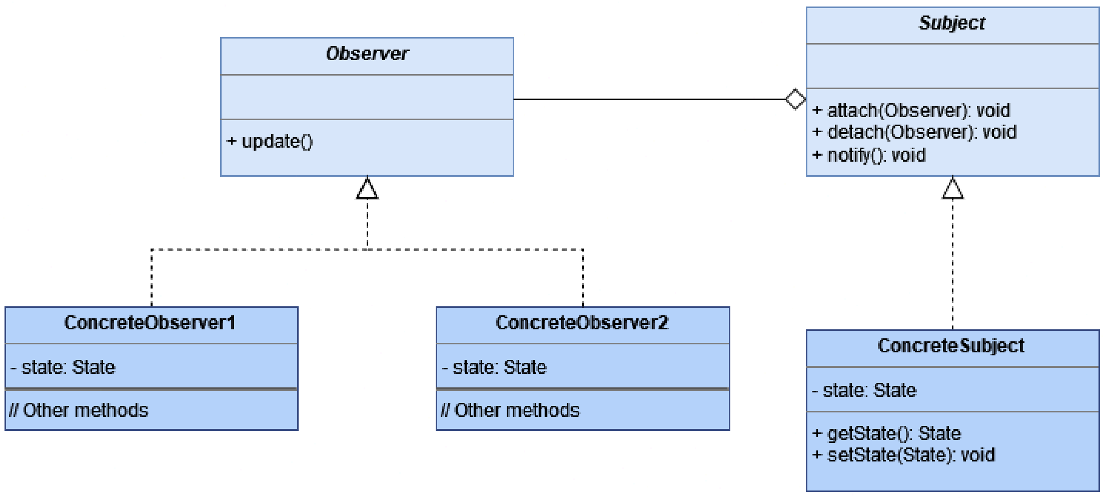
August 31st, 2024

INTRODUCTION:

The Observer design pattern is one of the most commonly used patterns nowadays. It is a behavioral pattern that defines a one-to-many dependency between objects, in which a change of state of one object is made known to other objects through a notification, allowing the observers to take the corresponding actions in response to the changes.
The observed object does not require any information about the observers, since the interaction is carried out independently through the observers' interface, which receives updates automatically.
The Observer pattern also improves in terms of scalability and provides an open-ended design that allows the addition of new observers without ever modifying the subject.

The Observer pattern implements the following elements:

- **Observable:** Also known as the Subject, it is an interface that declares the operations for adding or removing the observers from the subject.
- **Concrete Observable:** In charge of issuing events of interest to other objects when a change in its state occurs.
- **Observer:** Defines the update() method in which the appropriate actions should take place.
- **Concrete Observer:** Implements the necessary actions and changes in response to the changes made in the Subject.



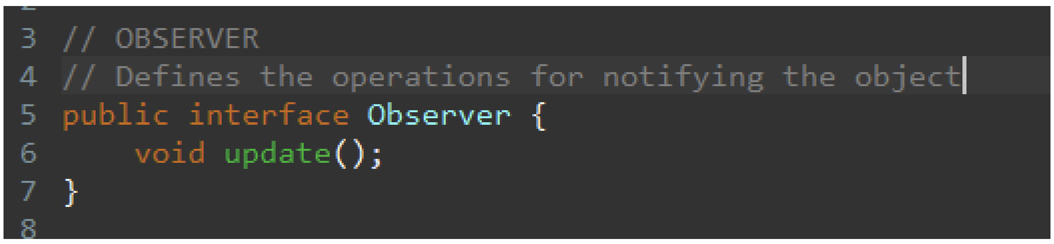
The observer pattern is used when a change of state in one object must be reflected in another object without keeping the objects tightly coupled, or when there's a need to implement a publish-subscribe kind of system.

For instance, in a digital collaboration platform, users of a particular team start assigning tasks to each other, each with important details to know about such as its title, description, priority level and due date.
Sometimes, unexpected obstacles and pending issues arise that forces the team to make changes on the status of a task. In that case, it is extremely important to let the changes be known to the members assigned to said task, so that they can re-evaluate their plans and have a chance to complete the task in a timely manner.

The use of an Observer pattern can be a viable solution for this problem, since it enables to send out important notifications to the assigned members about the new changes on a particular task.

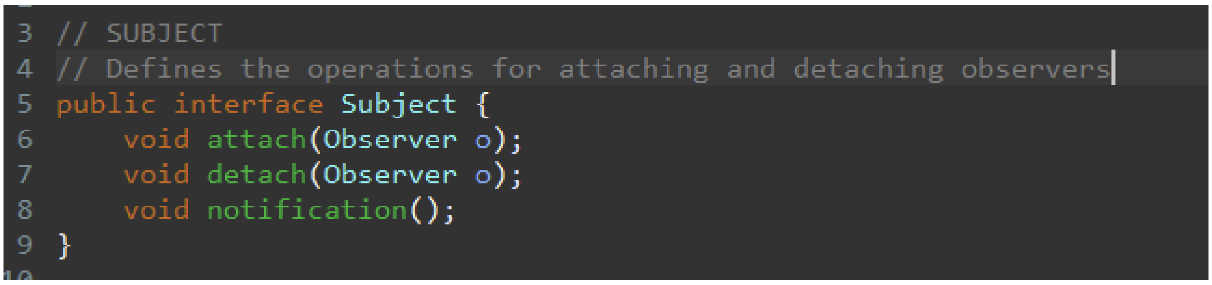
JAVA EXAMPLE:

Taking as an example the previous scenario of the notifications about the changes made in the team tasks, the basic components of the Observer pattern were developed to propose a simple solution, starting with the **Observer** interface that declares an interface of notification, consisting of a single update() method



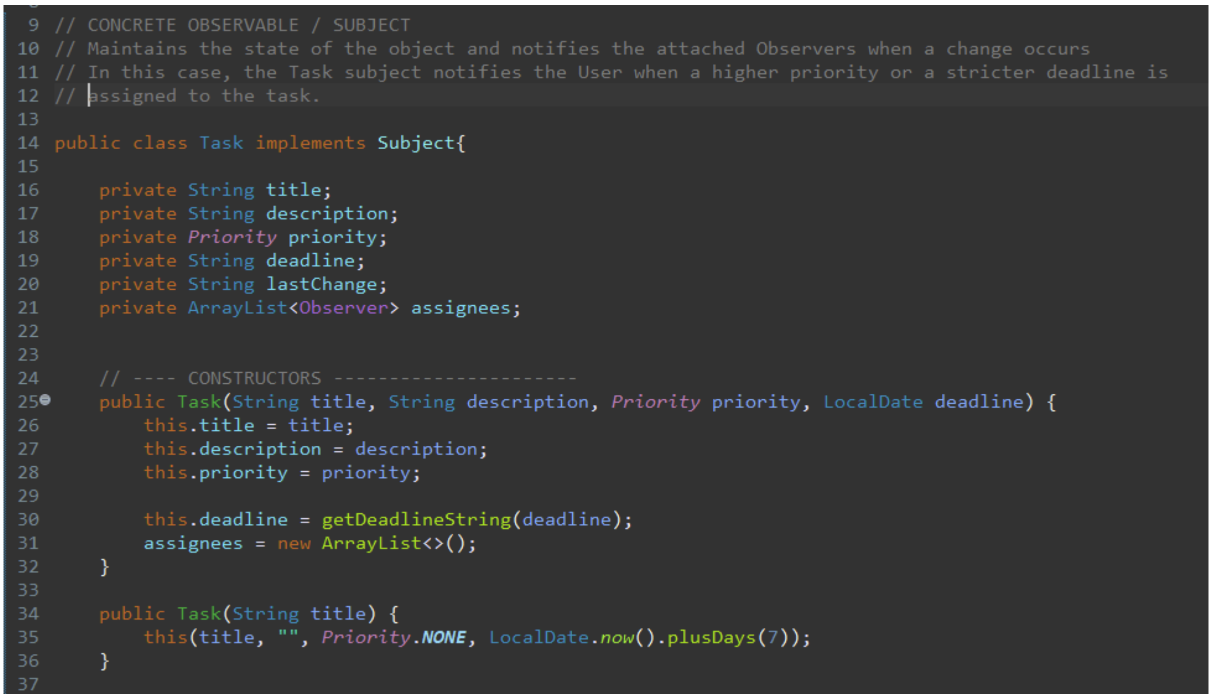
```java
// OBSERVER
// Defines the operations for notifying the object
public interface Observer {
    void update();
}
```

Similarly, the **Subject** (also known as the Observable) interface defines the operations for attaching and de-attaching observers, as well as a notification method to let observers know about the changes in state:

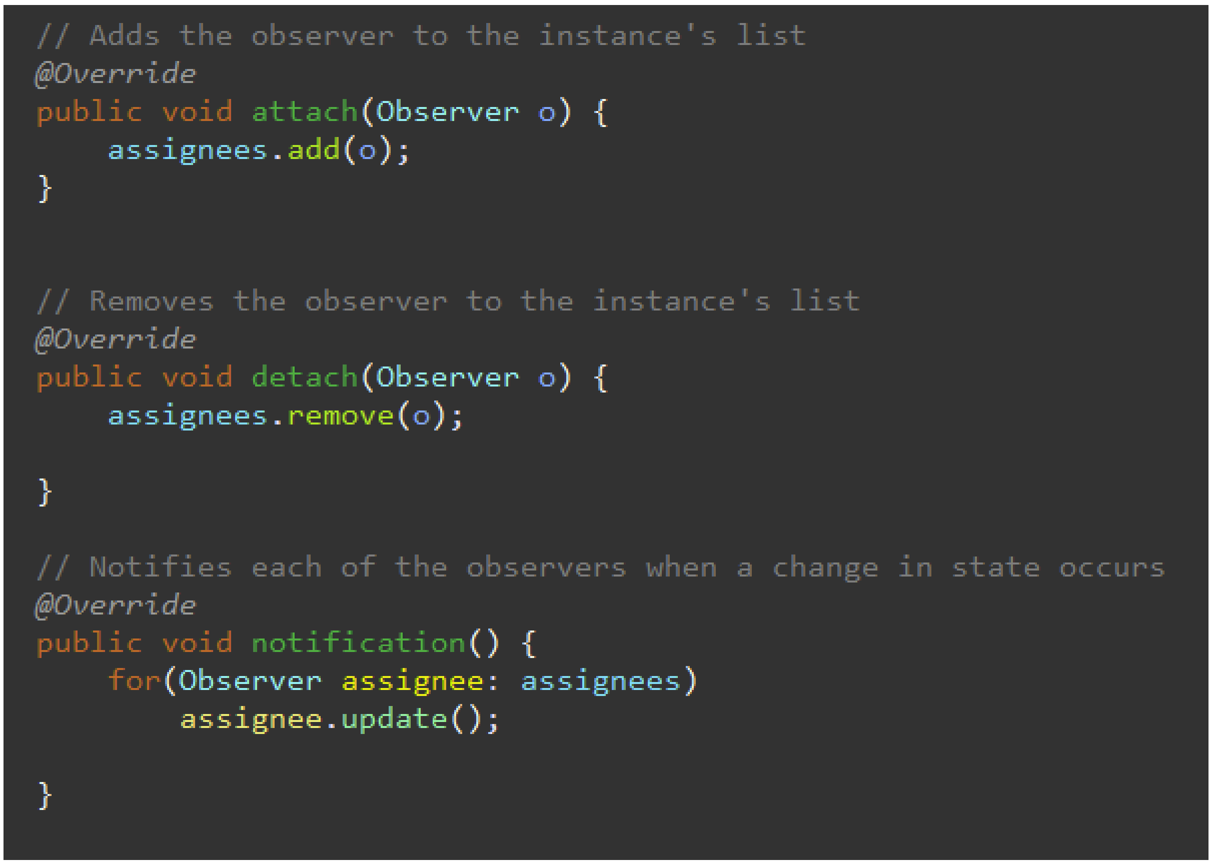


```java
// SUBJECT
// Defines the operations for attaching and detaching observers
public interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notification();
}
```

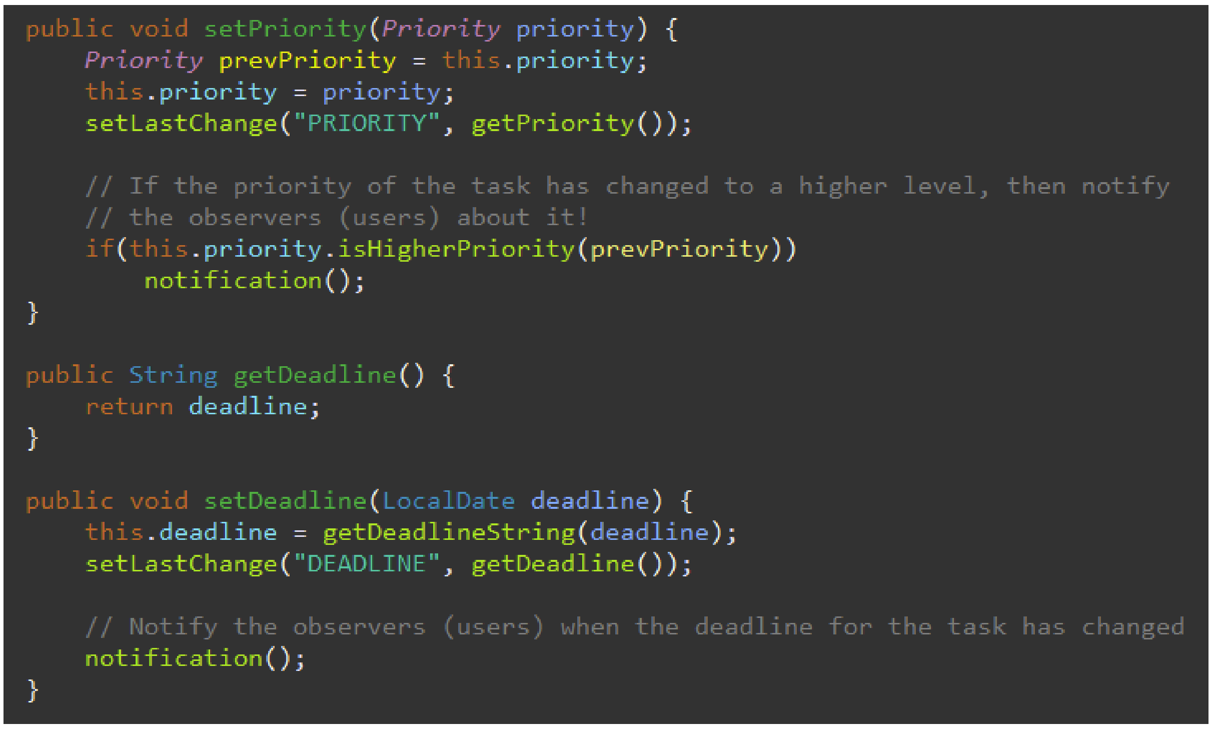
The **Task** class represents the **concrete Subject** that manages a list of observers attached to it, and implements the behavior for sending out events of interest after making particular changes to it. For this case, when the Task instance is assigned a higher priority than before, or when a new deadline is established, it notifies the observers from the list about it by calling the **update()** method included in the Observer interface:



```java
// CONCRETE OBSERVABLE / SUBJECT
// Maintains the state of the object and notifies the attached Observers when a change occurs
// In this case, the Task subject notifies the User when a higher priority or a stricter deadline is
// assigned to the task.

public class Task implements Subject{

    private String title;
    private String description;
    private Priority priority;
    private String deadline;
    private String lastChange;
    private ArrayList<Observer> assignees;


    // ---- CONSTRUCTORS ---------------------
    public Task(String title, String description, Priority priority, LocalDate deadline) {
        this.title = title;
        this.description = description;
        this.priority = priority;

        this.deadline = getDeadlineString(deadline);
        assignees = new ArrayList<>();
    }

    public Task(String title) {
        this(title, "", Priority.NONE, LocalDate.now().plusDays(7));
    }

```



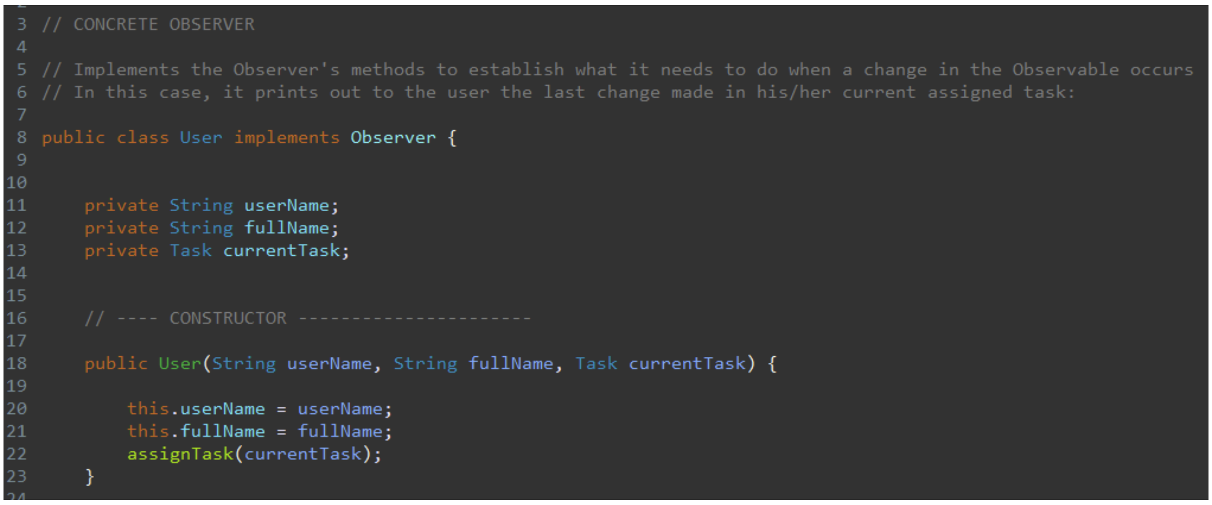
```java
// Adds the observer to the instance's list
@Override
public void attach(Observer o) {
    assignees.add(o);
}


// Removes the observer to the instance's list
@Override
public void detach(Observer o) {
    assignees.remove(o);

}

// Notifies each of the observers when a change in state occurs
@Override
public void notification() {
    for(Observer assignee: assignees)
        assignee.update();

}
```



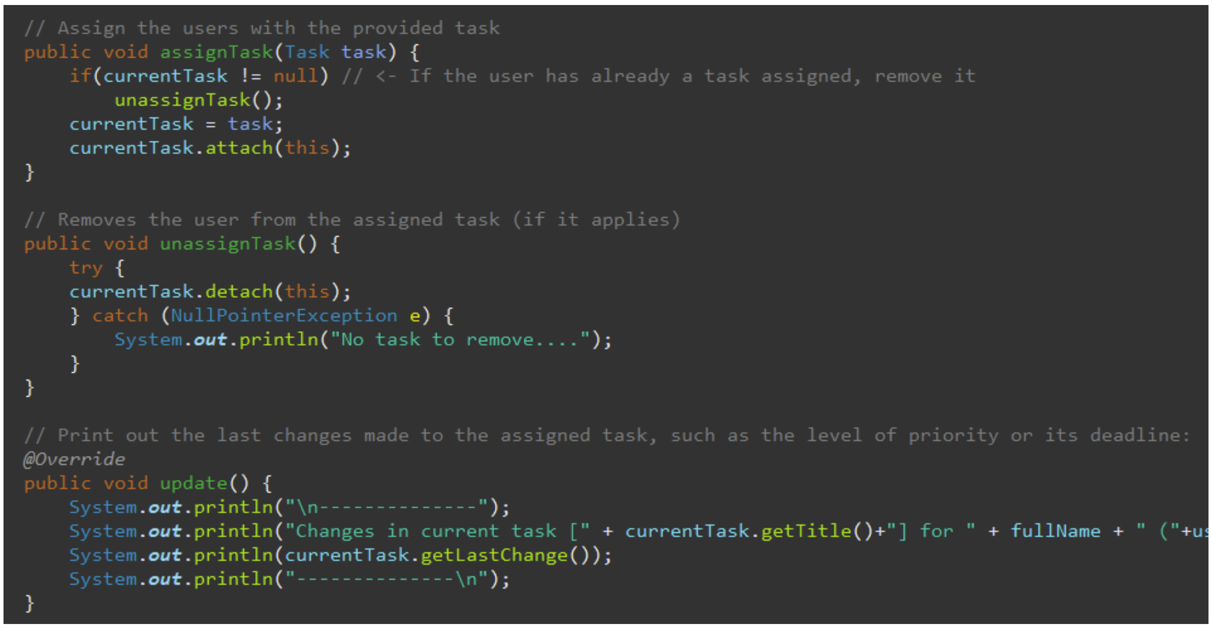
```java
public void setPriority(Priority priority) {
    Priority prevPriority = this.priority;
    this.priority = priority;
    setLastChange("PRIORITY", getPriority());

    // If the priority of the task has changed to a higher level, then notify
    // the observers (users) about it!
    if(this.priority.isHigherPriority(prevPriority))
        notification();
}

public String getDeadline() {
    return deadline;
}

public void setDeadline(LocalDate deadline) {
    this.deadline = getDeadlineString(deadline);
    setLastChange("DEADLINE", getDeadline());

    // Notify the observers (users) when the deadline for the task has changed
    notification();
}
```

On the other hand, the **User** class represents the **concrete Observer** that performs the appropriate actions in response to the notification sent to it by the Subject.

In this example, the update() method of the User only prints a message informing the user about the last change made to the task to which it is currently assigned to:



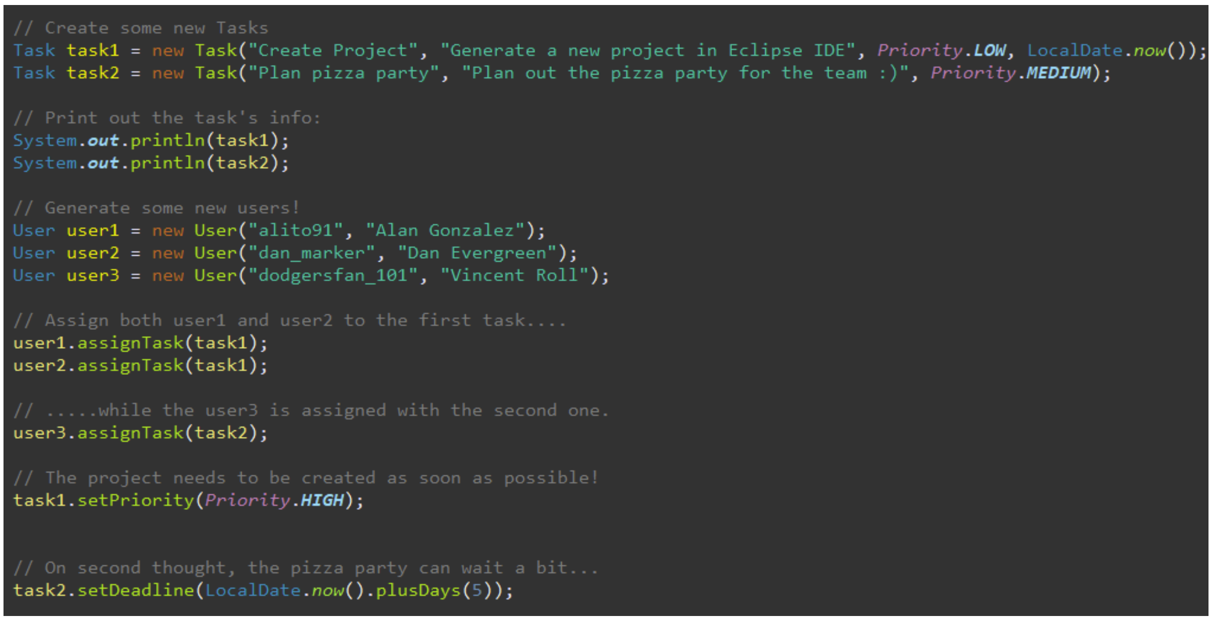
```java
// CONCRETE OBSERVER

// Implements the Observer's methods to establish what it needs to do when a change in the Observable occurs
// In this case, it prints out to the user the last change made in his/her current assigned task:

public class User implements Observer {


    private String userName;
    private String fullName;
    private Task currentTask;


    // ---- CONSTRUCTOR ----------------------

    public User(String userName, String fullName, Task currentTask) {

        this.userName = userName;
        this.fullName = fullName;
        assignTask(currentTask);
    }
```



```java
// Assign the users with the provided task
public void assignTask(Task task) {
    if(currentTask != null) // <- If the user has already a task assigned, remove it
        unassignTask();
    currentTask = task;
    currentTask.attach(this);
}

// Removes the user from the assigned task (if it applies)
public void unassignTask() {
    try {
    currentTask.detach(this);
    } catch (NullPointerException e) {
        System.out.println("No task to remove....");
    }
}

// Print out the last changes made to the assigned task, such as the level of priority or its deadline:
@Override
public void update() {
    System.out.println("\n-------------");
    System.out.println("Changes in current task [" + currentTask.getTitle()+"] for " + fullName + " ("+us
    System.out.println(currentTask.getLastChange());
    System.out.println("-------------\n");
}
```

To test the application of the Observer pattern, a duplicate of tasks with different properties is generated and printed out, to then generate a group of users that could be part of a team.
user1 and **user2** are assigned to the first task of creating the project, while **user3** is assigned to the task of organizing a pizza party.

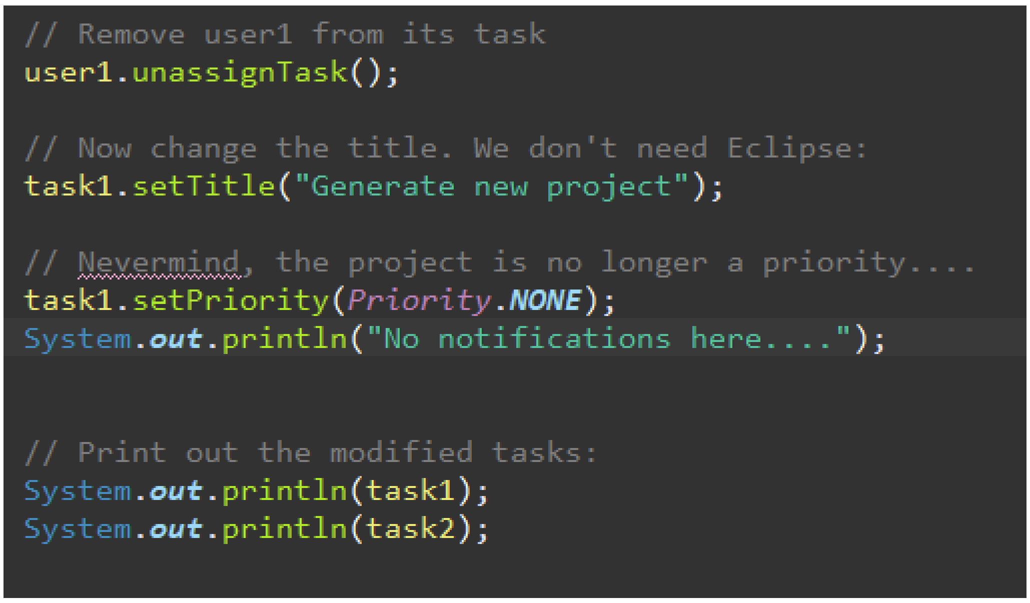
This is where the priority of the first task is changed to high, while the deadline of the second task is changed, which should prompt the corresponding notifications to each user.



```java
// Create some new Tasks
Task task1 = new Task("Create Project", "Generate a new project in Eclipse IDE", Priority.LOW, LocalDate.now());
Task task2 = new Task("Plan pizza party", "Plan out the pizza party for the team :)", Priority.MEDIUM);

// Print out the task's info:
System.out.println(task1);
System.out.println(task2);

// Generate some new users!
User user1 = new User("alito91", "Alan Gonzalez");
User user2 = new User("dan_marker", "Dan Evergreen");
User user3 = new User("dodgersfan_101", "Vincent Roll");

// Assign both user1 and user2 to the first task....
user1.assignTask(task1);
user2.assignTask(task1);

// .....while the user3 is assigned with the second one.
user3.assignTask(task2);

// The project needs to be created as soon as possible!
task1.setPriority(Priority.HIGH);


// On second thought, the pizza party can wait a bit...
task2.setDeadline(LocalDate.now().plusDays(5));
```

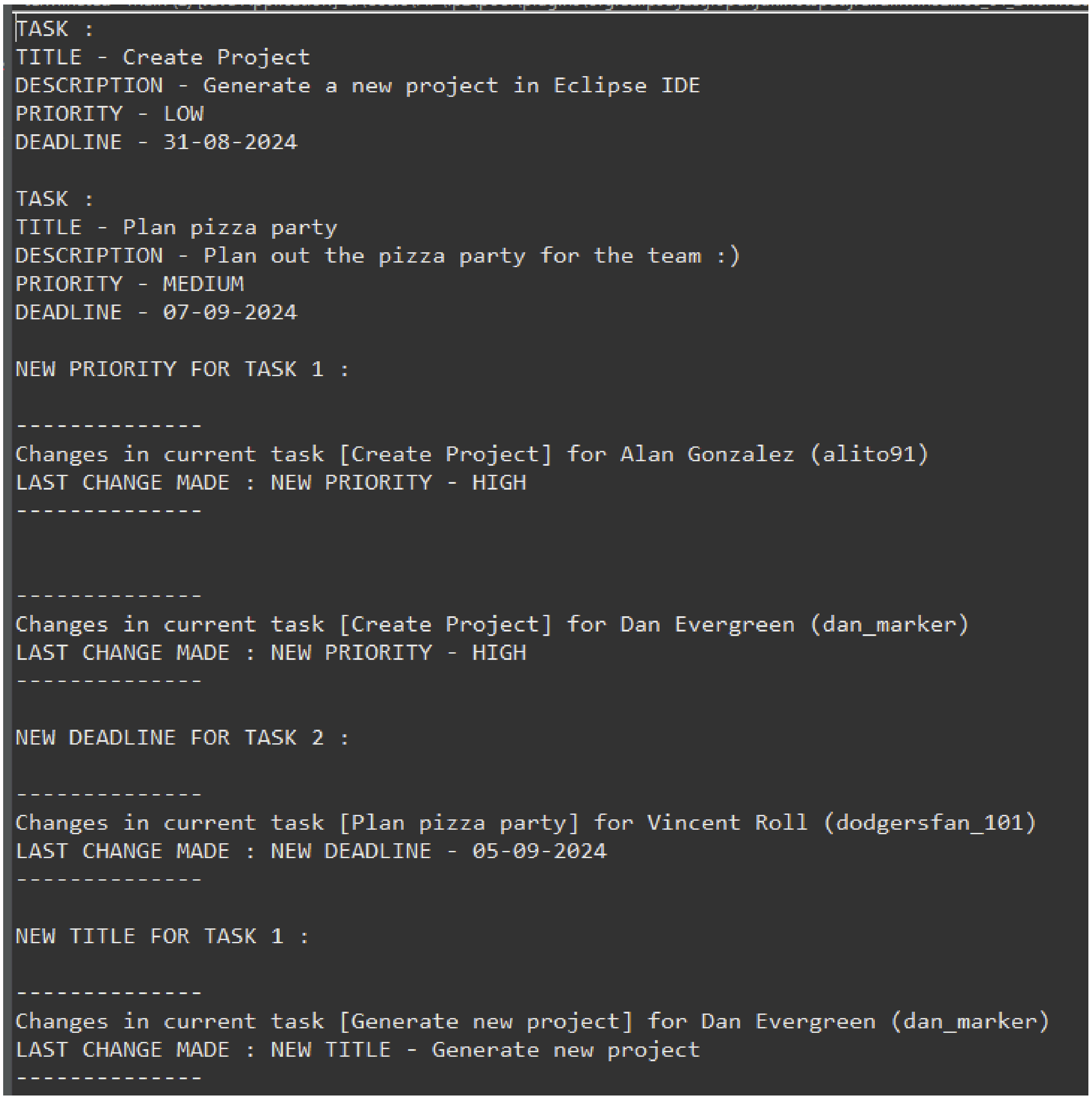
user1 is removed from its task. As such, the notification about the new change for the first task, such as a change in its title, should only be sent out to user2.

When a new priority for a task is lower than the previous one, then no notification will be sent out to the users.
Lastly, the modified tasks are printed out:

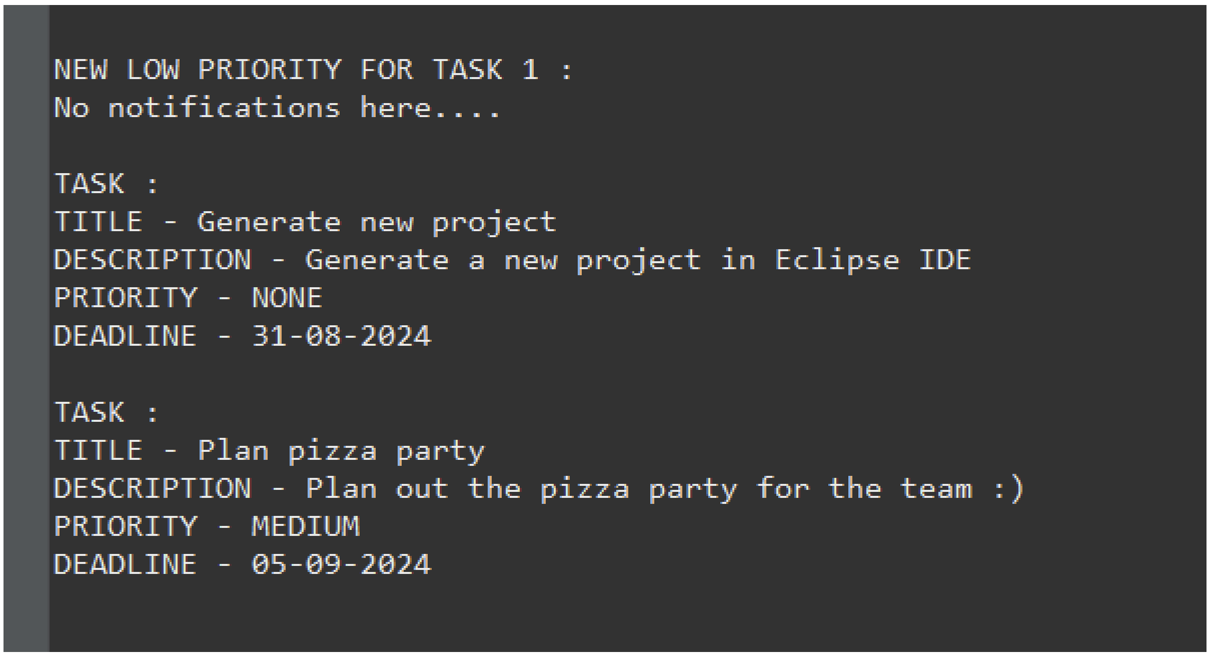


```java
// Remove user1 from its task
user1.unassignTask();

// Now change the title. We don't need Eclipse:
task1.setTitle("Generate new project");

// Nevermind, the project is no longer a priority....
task1.setPriority(Priority.NONE);
System.out.println("No notifications here....");


// Print out the modified tasks:
System.out.println(task1);
System.out.println(task2);
```

RESULT:



```
TASK :
TITLE - Create Project
DESCRIPTION - Generate a new project in Eclipse IDE
PRIORITY - LOW
DEADLINE - 31-08-2024

TASK :
TITLE - Plan pizza party
DESCRIPTION - Plan out the pizza party for the team :)
PRIORITY - MEDIUM
DEADLINE - 07-09-2024

NEW PRIORITY FOR TASK 1 :

--------------
Changes in current task [Create Project] for Alan Gonzalez (alito91)
LAST CHANGE MADE : NEW PRIORITY - HIGH
--------------



--------------
Changes in current task [Create Project] for Dan Evergreen (dan_marker)
LAST CHANGE MADE : NEW PRIORITY - HIGH
--------------

NEW DEADLINE FOR TASK 2 :

--------------
Changes in current task [Plan pizza party] for Vincent Roll (dodgersfan_101)
LAST CHANGE MADE : NEW DEADLINE - 05-09-2024
--------------

NEW TITLE FOR TASK 1 :

--------------
Changes in current task [Generate new project] for Dan Evergreen (dan_marker)
LAST CHANGE MADE : NEW TITLE - Generate new project
--------------
```



```
NEW LOW PRIORITY FOR TASK 1 :
No notifications here....

TASK :
TITLE - Generate new project
DESCRIPTION - Generate a new project in Eclipse IDE
PRIORITY - NONE
DEADLINE - 31-08-2024

TASK :
TITLE - Plan pizza party
DESCRIPTION - Plan out the pizza party for the team :)
PRIORITY - MEDIUM
DEADLINE - 05-09-2024
```

The use of the Observer pattern allows for a simple yet effective solution that communicates the changes from one object to multiple observers, as was the case with notifications to users about changes in the assigned task, allowing them to take appropriate actions to adapt to said changes and stay up to date.