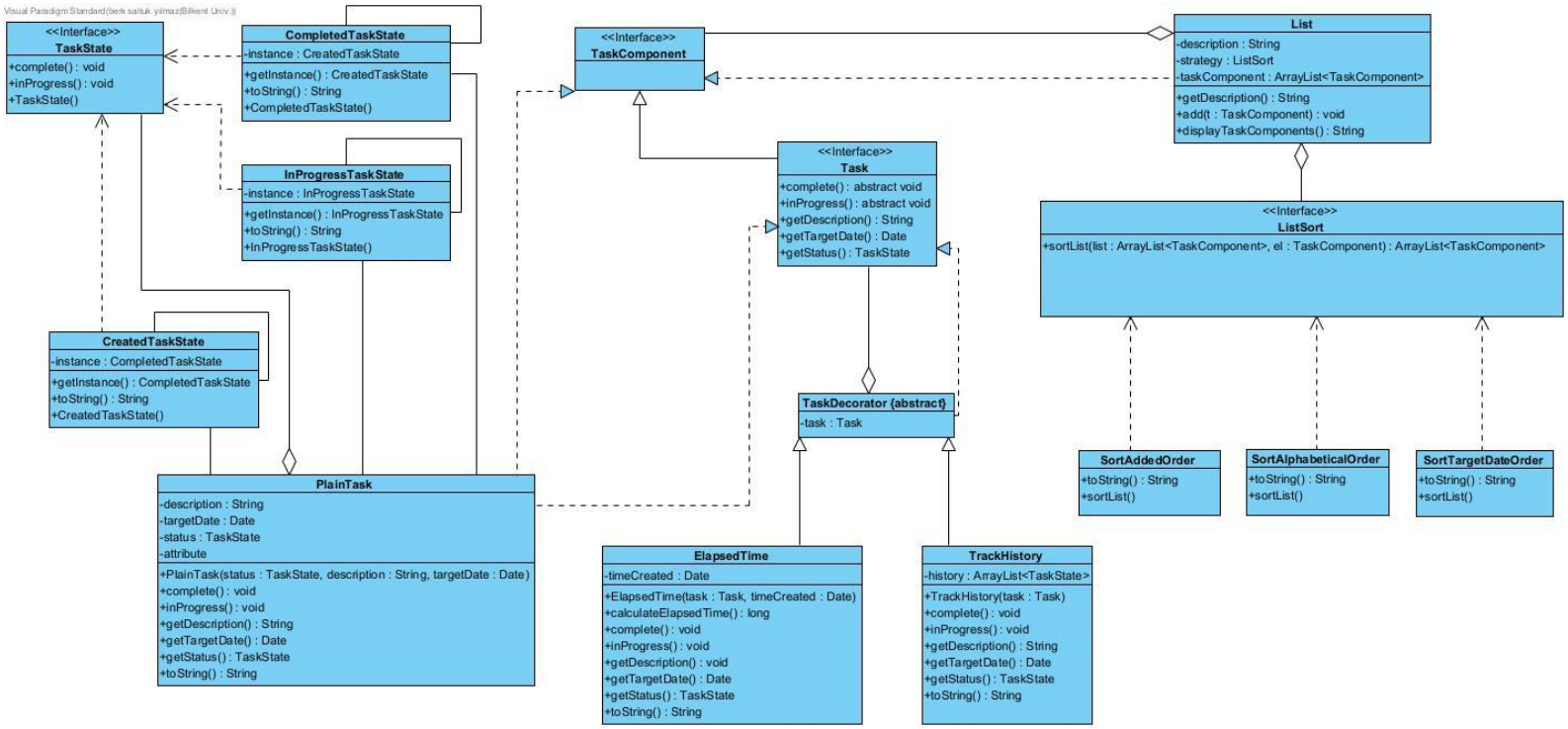
1. UML Diagram of Implementation



(Figure 1: UML Diagram for Implementation)
(For high-resolution image: https://www.hizliresim.com/m2qcmq7)

2. Patterns Used in Implementation

I have used State Pattern, Decorator Pattern, Composite Pattern, and Singleton Pattern, Strategy Pattern in this implementation.

2.1 State Pattern

Classes involved: CompletedTaskState, InProgressTaskState, CreatedTaskState, PlainTask
Interfaces involved: TaskState.

State Pattern was used for internal transitions of the task's status property. For this purpose, an interface that is implemented by each state class was introduced. An instance of this interface was added to PlainTask class which is the context of state and transitions. With this design pattern, all the transitions were done on the state objects, and a finite state machine-like structure was obtained, and as a result, complexity was reduced.

2.2 Singleton Pattern

Classes involved: CompletedTaskState, InProgressTaskState, CreatedTaskState

Singleton Pattern was also used in the State classes that implement the TaskState interface. With Singleton Pattern, an instance only gets initialized if and only if it is needed; that is, if no one invokes the getInstance method, the State object

will not be initialized. Moreover, with Singleton, it is guaranteed that there will be at most one instance for each state. This pattern provides better control of the instances.

2.3 Decorator Pattern

Classes involved: TaskDecorator (abstract), ElapsedTime, TrackHistory, PlainTask
Interfaces involved: Task.

Decorator Pattern was used for two different options that come with Tasks. It would be an overhead to create classes such as TaskWithElapsedAndTrackHistory; therefore, an abstract decorator class that extends the Task component was introduced. Also, new behaviors were added by extending the base decorator. This pattern allows adding new behaviors dynamically to the Task objects without affecting others.

2.4 Composite Pattern

Classes involved: List, PlainTask
Interfaces involved: TaskComponent, Task

Since each list can contain lists or tasks, I was reminded of the tree-like directory-file example we discussed in class. I thought the same tree example could be observed in this list case; therefore, I have applied the Composite Pattern by introducing a TastComponent and implementing this TaskComponent in List and the leaf which is PlainTask. Moreover, I have added an ArrayList of TaskComponents to the List class to make tree structure possible. This pattern allowed me to have nested lists and tasks as leaves. I have also made Task to extend TaskComponent to make the decorator pattern possible since each decorative behavior must also be compatible to be added to the list.

2.5 Strategy Pattern

Classes involved: List, SortAddedOrder, SortTargetDateOrder, SortAlphabeticalOrder
Interfaces involved: ListSort

Strategy Pattern was used because different sorting strategies were used for other list-objects. Therefore ListSort interface was introduced and implemented by different sorting strategies. This pattern was the best pattern to implement different sorting techniques because the list objects were only differing by sorting algorithm used. This pattern also provided flexibility to change the sorting strategy dynamically since List has the strategy interface as an instance. It can be initialized as any of the strategies that implement the interface.