Steven Mayo
CSC 216
Project 2 Part 1
Section 601

March 17, 2017

ScrumBacklog Design Proposal

1. DESIGN OVERVIEW

To fully implement the requested behavior of the client, a design proposal of ten classes have been put forth and can be seen in Figure 1. The swing components and other behaviors relating to the user interface will be contained in the `ScrumBacklogUI` class. Although the user will be interacting with this class for user input, the program's core logic will primarily be housed in the `ScrumBacklog` class, itself. The `ScrumBacklog` class utilizes the State Pattern approach for handing how a task will be processed as it is completed.

To implement the State Pattern design approach, the abstract `State` class puts forth common behaviors that each `State` object should have access to. In essence, the client needs to be able to perform 3 basic actions on a task at all times: process/advance the task to the next state, return the state to the backlog, or simply simply reject the task. Therefore, these methods were declared abstract within the `State` class and left for each `State` object to tailor the behaviors to their own specific needs.

In terms of the `State` objects themselves, a total of six classes are used to fully implement the `State` Pattern design approach: `BacklogState, OwnedState, ProcessState, VerifyState, DoneState`, and `RejectState.` Each of these six classes inherit common behavior and attributes from the `State` class, while also serving as composition classes for `ScrumBacklog`**.** Furthermore, these classes were chosen to be nested within the `ScrumBacklog` class itself, since `State` functionality does not need to extend beyond that of the `ScrumBacklog` class. Additionally, because these classes were nested, they still retain access to private members and behaviors that are specific to `ScrumBacklog`, but applicable to state transitions and `Task` processing.

Generating `Task`s, as well as processing attributes related to task management, such as file input, file output, task list generation, note generation, etc… is not shown in Figure 1, since it will be contained within a separate package that was provided by the client. However, this package is extremely important and critical for ensuring proper backlog functionality. In essence, as the user interacts with the `ScrumBacklogUI` class, an instance of `ScrumBacklog` will utilize the `Task XML Library` to generate, edit, and/or delete `Task` Objects, or the `TaskList` containers themselves. Additionally, both the Note and `NoteList` objects will be handled by the `Task XML Library`, which will be relayed back to the user via `ScrumBacklogUI` .

Lastly, an `InvalidTransitionException` class, seen in Figure 1, is being prosed as an additional check for proper control flow and state transitioning. Aside from the checks and error handling defined within the `Task XML Library`, this class will be used to throw a checked exception for any unpredicted or invalid state transitions.

2. CONTROL FLOW OVERVIEW

The initial flow of control begins at the `ScrumBacklogUI` class, where the `main()` function is called to produce an instance of the class itself and allow for user input. After the user interface is established, the user has the ability to create a new task file, load a task file, save a task file, or quit the application. Selecting "Load", "New", or "Save" will trigger the `ScrumBacklogUI` class to call on file input, file output, and task generation behaviors defined within the `Task XML Library`. If a file is needed for input, then an instance of `TaskReader` is instantiated with a filename parameter to handle required parsing. Likewise, `TaskWriter` will be instantiated with a String parameter for the filename and a `TaskList` object to output the backlog of `Task` objects. Selecting "Quit" will simply terminate the program.

Once a `TaskList` has been populated - via `Task XML Library` file handling behavior - or manually entered - via exchange between input from the UI class and task generation behavior from the the `Task XML Library` package – control will be directed to the `ScrumBacklog` class. This class was primarily designed to handle state transitions, which correspond to how a task can be processed as it undergoes completion.

The flow of control switches from the user interface to the `ScrumBacklog` class as soon as a task is added to the backlog. At this time, the task enters the `BacklogState`, where it will be assigned a unique identification number and wait in queue for an owner. From this state, the `Task` can either transition to the `OwnedState` once selected by a user, or transition to `RejectState` if the project lies outside of the team's scope. From the `OwnedState`, a `Task` will transition to the `ProcessState` once the owner begins work on the task. Otherwise, the owner can place the `Task` back into the `BacklogState` - for queue - or into the `RejectState`. From `ProcessState`, the `Task` can transition to `VerifyState`, where a `Task` must wait in queue for verification from a team member, before either being sent back into `ProcessState` for improvement, or sent to the `DoneState`. At the `DoneState`, a task has the ability to transition back into the `ProcessState` or back into the `BacklogState`. Any `Task` that has been marked for rejection will fall under the `RejectState` class control. If rejected, the `Task` will either remain listed as rejected or be sent back into the backlog, where the `BacklogState` class will dictate control over progression.

Again, these states can be reached via three basic approaches of task handling: progressing backwards, progressing forwards, or rejection. Therefore, as dictated by the abstract State class, backlogTask(), processTask(), and rejectTask() are implemented as needed, respectfully. The main difference between classes of the proposed State Pattern design is with how each independently defines the local processTask(). For example, if a task is within the BacklogState, then processTask() is used to transition into the OwnedState by setting the task's state label, logging the note label, and setting the task's owner label. Yet, the definition of VerifyState.processTask() has the additional behavior of assigning the verifier's id. These differences are what control the flow of a Task's progression through ScrumBacklog and are essentially what differentiates one State from another.

3. LIMITATIONS AND CONSTRAINTS OF THE PROPOSED DESIGN

There is a slight limitation with the proposed design, as well as, several constraints placed upon the system. However, these limitations were taken into consideration and will not significantly impact the overall functionality of the program. Additionally, constraints are present and/or implemented to conform to requirements requested by the client.

The first limitation of this design is with the implementation of ScrumBacklog If at any time in the future the client needs another state transition added, then it must be hard coded and housed within the ScrumBacklog class. Additionally, this class already has 8 nested classes, making both the file size and file readability less than ideal. Furthermore, having such a large overhead of code within one class places a significant burden of on the Runner thread. If stressed too much, the program or system running the program can become unresponsive or crash.

In terms of constraints, this design does not allow for a Task to be removed once placed into the backlog. A task can go into the RejectState, but cannot be deleted entirely from the backlog. This could also be considered a limitation, but it is behavior specifically requested by the client. Additionally, this design constrains the ownership of a task by allowing only one assignment to the task owner. Lastly, this design constrains task progression by requiring verification before proceeding to the DoneState. However, doing so is beneficial and leads to an overall increase in code reliability.

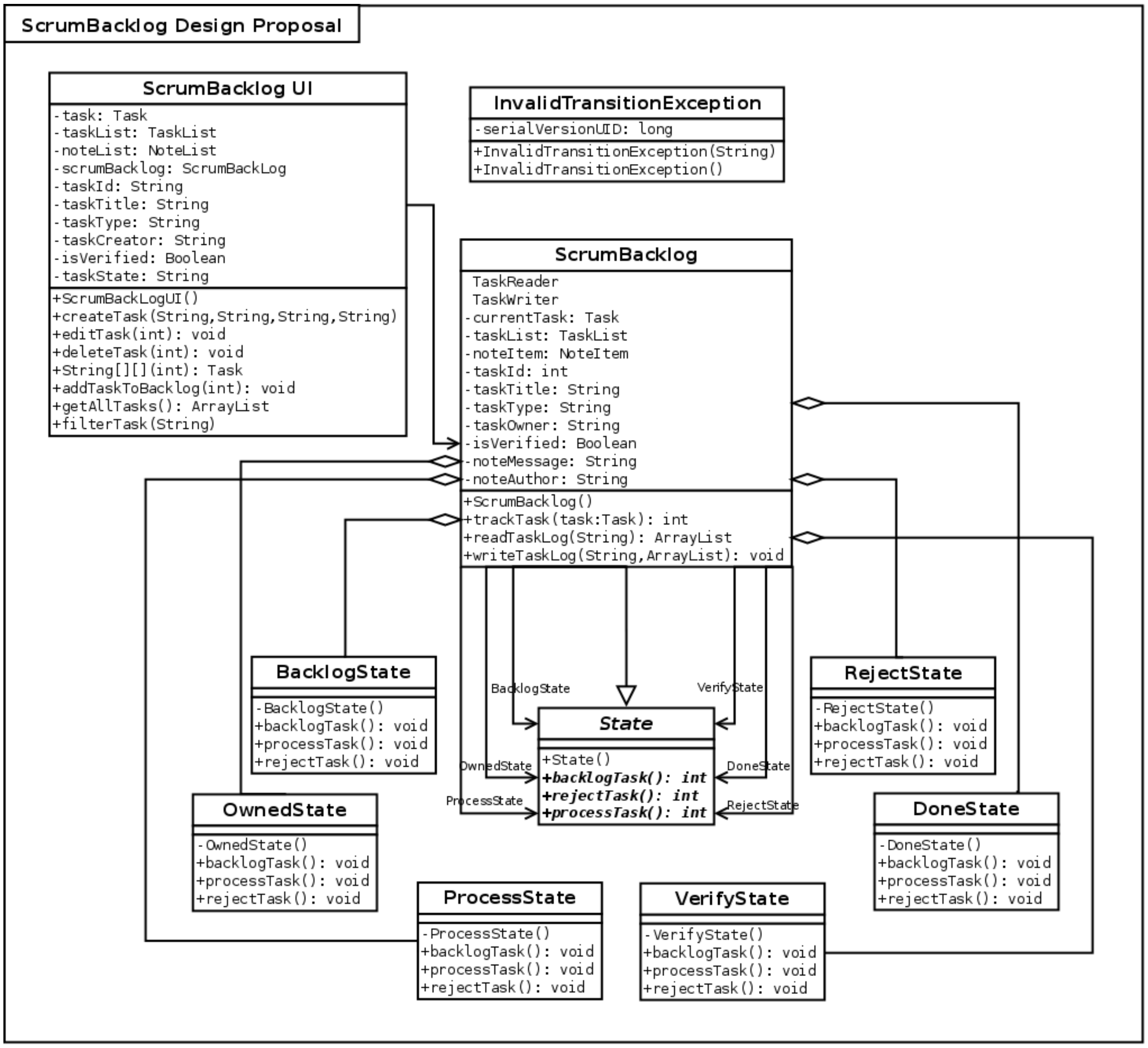

Figure 1: Proposed UML Diagram for ScrumBacklog Application.