Thomas Finn Lidbetter and Patrick Coyle
Comp 3721
Dr. Ricker
TAC Milestone 4
November 18, 2016

TAC Documentation

Reasons for selecting TAC:

We chose to use Finn and Michael's TickAttack project largely because this already made use of more design patterns than Patrick and Lucas' game. Furthermore TAC seemed to be better organised than the alternative and more extendable in most aspects. We did, however, find that adding a new quest was more difficult than ti might have been in Patrick and Lucas' implementation. More details on this will be given in the section on quest incorporation. In addition to this, while we had possible new structural design patterns in mind for both implementations we found that the decorator pattern implementation in TAC was more relevant and impactful to the overall design than any structural design pattern that we could conceive for Patrick and Lucas' game.

Quest Insertion Process:

As mentionned in our discussion of the choice for TAC, the quest implementation was a bit rockier than expected. The quest I decided to add was the "Surf Internet" quest from my original game, which is largely a dialog based quest, with the ability to choose how much time we wait between each line of dialog, and to receive rewards. In general, since TAC has a lot of randomness basically embedded into the code, and that the quest I have has no randomness, it was a little difficult sometimes to force things to be deterministic, as will be explained in the process of incorporating the quest. This is in opposition to my game, where quest modifications would have been relatively simple due to the read file format of the quests. Indeed, we were aware quests would be easier to begin with on the alternative Tick Attack, but as mentioned in the TAC discussion, it appeared to be not much more complicated, and provided a more interesting application of a structural design pattern.

Evidently, I began by creating the SurfInternetQuest.java file, which consists of the Surf Internet class. I started by creating a constructor, which due to the design of TAC, forces me to call the super constructor. This unfortunately has the domino effect of setting the time it takes to complete quest to something essentially random, which is problematic in a dialog based quest that makes more sense to end soon after the dialog ends, and not before or long after. So, I had to set the "timeToComplete" variable of the quest to an appropriate value directly after the super call, so that we would not have the issue that the timer runs out before or long after the dialog finishes.

After trying a few things, I realized that it was also difficult to do things while the game is counting down the quest. There was no already present way to have dialog go off

within the time limit of a quest: as it was, updates to the view were restricted to before and after. To solve these problems, I went to the controller.java file, and changed the "attemptStartQuest" to return a boolean. I did it this way so that it wouldn't change any existing behaviour, but I could now use it to check whether it's okay to start the quest. If true, in the "process(String)" method, I call a new method called "startDialog", which starts a dialog, and add it to the Quest abstract class, to enable other quests to utilize the dialog option.

Finally, for the actual dialog Quest functionality, I added a few methods to the Quest Abstract class, and not to the SurfInternetQuest.java file, since it is preferable to extend quest rather than limit the functionality to a single quest, since we may want to reuse the code and/or extend it, and it won't affect existing quests since the added methods are not abstract methods and are defined generally, so there is no need to override them in other quests, nor implement them. The first method in question is getQuestFile, which gets the file stored in the questFiles directory whose name starts with the questName variable and ends with ".tad", which for now stands for "Tick Attack Dialog", but could be changed in the future to .taq for "Tick Attack Quests", to cover more ground. The second method is "readFile", which calls getQuestFile and uses the resulting buffered reader and stores it in a class variable. It then creates a timer that, every delay, will call "readDialog", the third and last method, which reads the dialog and sends to the view through the controller. The readDialog then returns the second string as a number, and tells the timer that the next delay will be that many seconds. This way, as in my version of Tick Attack, we can control the time a certain dialog stays on screen. If these methods are not used (as is the case with the original rangerQuest and fishingQuest), they remain unused. I designed it this way to adhere to the open-closed principle since, as mentioned, I am simply extending quest and a few functions, and am leaving it open for further extension in a similar fashion to the alternative Tick Attack by Me and Lucas, since it reads in files, and we could interpret particular syntax to do different things within the program. Thus, decision branches, randomness, and other things could be implemented through this.Finally, I noticed that going on your computer still had a chance to net a tick, which didn't make much sense. To fix this, I made the "hasTick" variable protected instead of private, so that subclasses (my quest) could change the value, and changed the hasTick = false, so as to ensure no ticks will be had.

In summary, the quest ended up working fairly well, though it was a little bit difficult to implement. However, TAC was open to modification, and I was able to find a few places to change some code without affecting existing functionality in any way, well successfully adding my quest as well as adding a little functionality to futures quests. Overall, the quest insertion went very well.

Design Pattern Justification and Incorporation:

We chose to incorporate the decorator design pattern into TAC. We observed that when a player buys fishing rods and ranger gear, similar behaviour occurs for each item added. The fishing skill or ranger skill is incremented and the best ranger gear or best fishing rod is set. In each case we can think of these additions and modifications as being 'decorations' for the player. As such we can eliminate the need for the fishingSkill and rangerSkill int variables,

and the bestRod and bestGear variables in the player. These can then be replaced with references to the classes that are being decorated - the BaseFishingSkill and BaseRangerSkill classes.

Incorporating this design pattern required very little modification of the existing code. The Player class had to be adjusted to remove the old fishingSkill, rangerSkill, bestRod, and bestGear variables. The functionalities associated with the old variables in the Player class then had to be modified in order to interact with the new FishingSkill and RangerSkill classes. The only other place in the existing code that then had to be modified was the set of lines in the Controller class that incremented the fishingSkill and rangerSkill variables when the Player purchases a new rod or new gear. The addition of the Decorator pattern also took advantage of the existing implementation. It made use of the FishingRod and RangerGear enums in order to get all of the information necessary for the FishingSkill and RangerSkill updates. This meant that all different rods and gear with which the skills can be decorated can be given by the 'AddRod' and 'AddGear' classes. Due to the similarities in behaviour of each of rods and gear, this solution seemed much cleaner than creating separate concrete components for each type of rod and each type of ranger gear.

Overall the process of incorporating this structural design pattern required little modification to the existing code. Going forward, it also provides an easy means to add new FishingRods, RangerGear, or other items that affect the Fishing and Ranger Skills. In these respects it adheres to the open-closed principle.