WCC - W19

CS240

Professor Ryan Parsons

Authors:

Adib Thaqif

Andrew Jacobi

Micah Miller

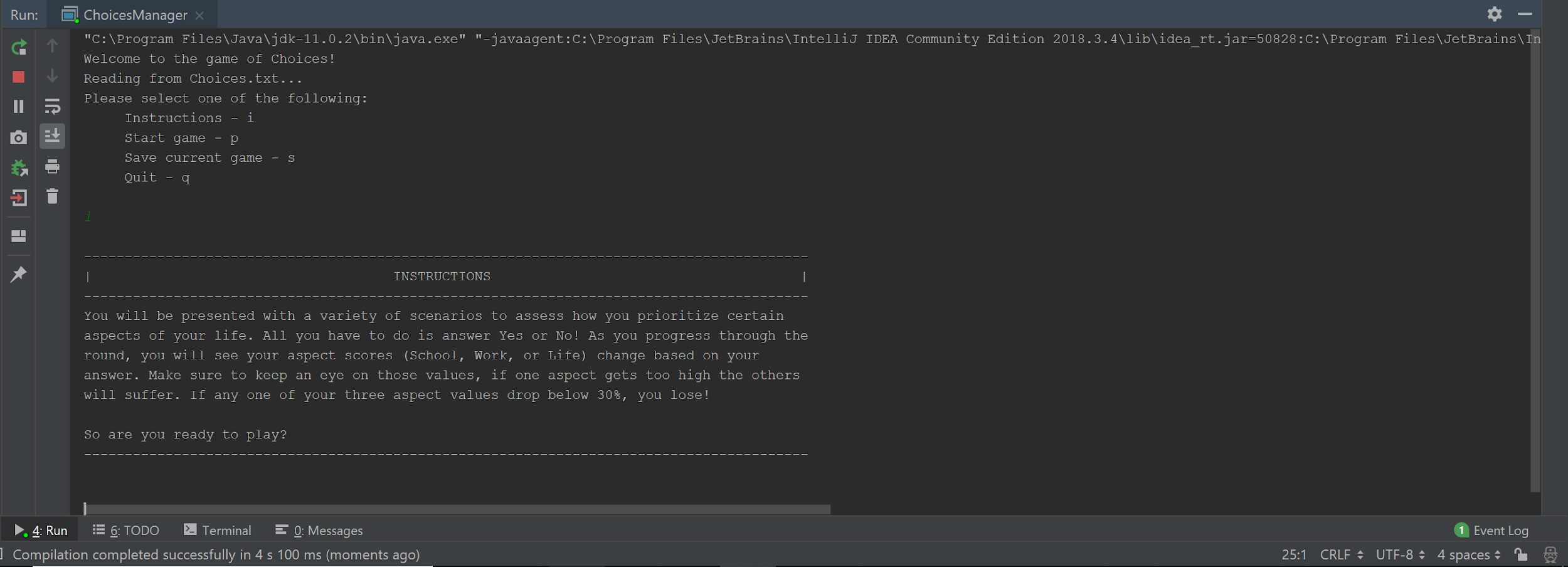
Donald Strong

A Game of Choices

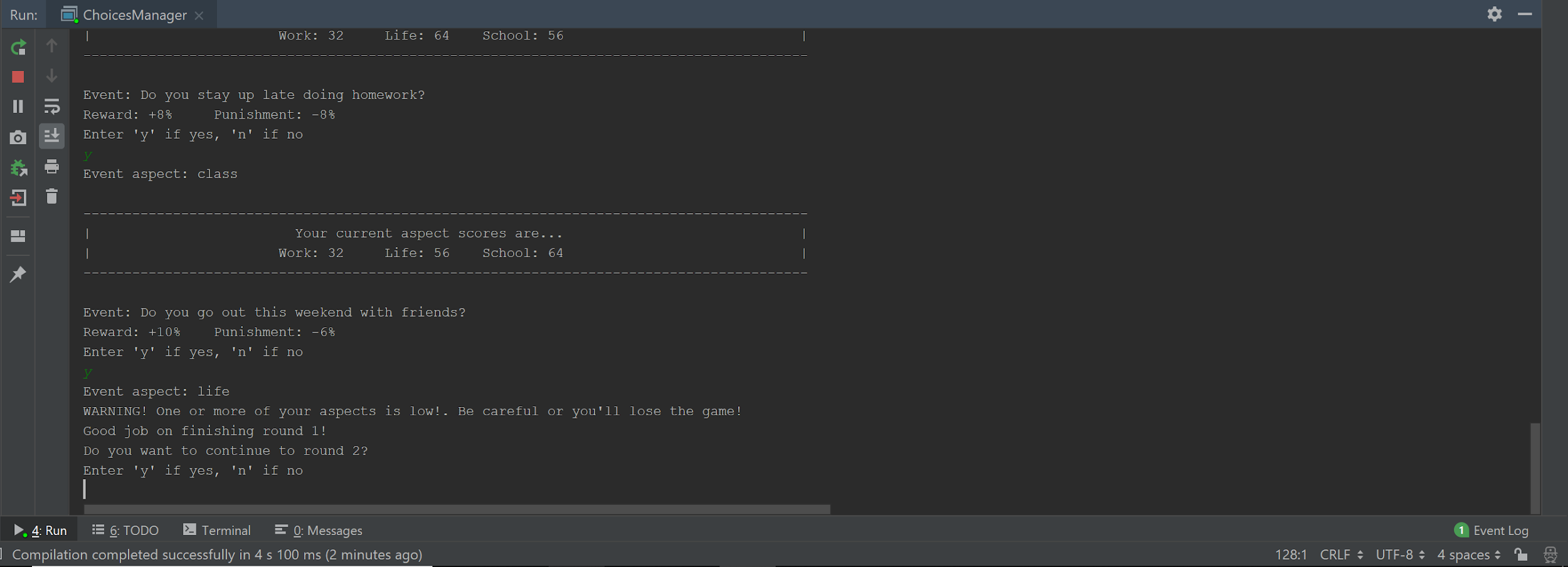
**Introduction / Demonstration**

For our quarter-long project, we elected to create a simple game: a game of choices. The game is designed to present the user with a variety of scenarios to assess which aspects of their life they prioritize the most. The three aspects that are covered in the game include: work, life (social), and school. Each question will give the user a yes and no option for which the user will have to determine what they’d like to choose. With each answer comes a reward and punishment system that affects each aspect. The user’s goal is to maintain a certain threshold and make it through the lines of questioning that are designed to simulate all four years of college. If the user succeeds, they win the game! Otherwise, they lose and will have to start over.

*Example of the Main Menu and Instructions*



We successfully completed a program that asks the user a series of questions for each round. As the user advances through the rounds, the lines of questioning will get longer and increase the difficulty to maintain the threshold. As previously mentioned, the game is over if one of the aspect scores falls below a certain threshold (the default value is 25).

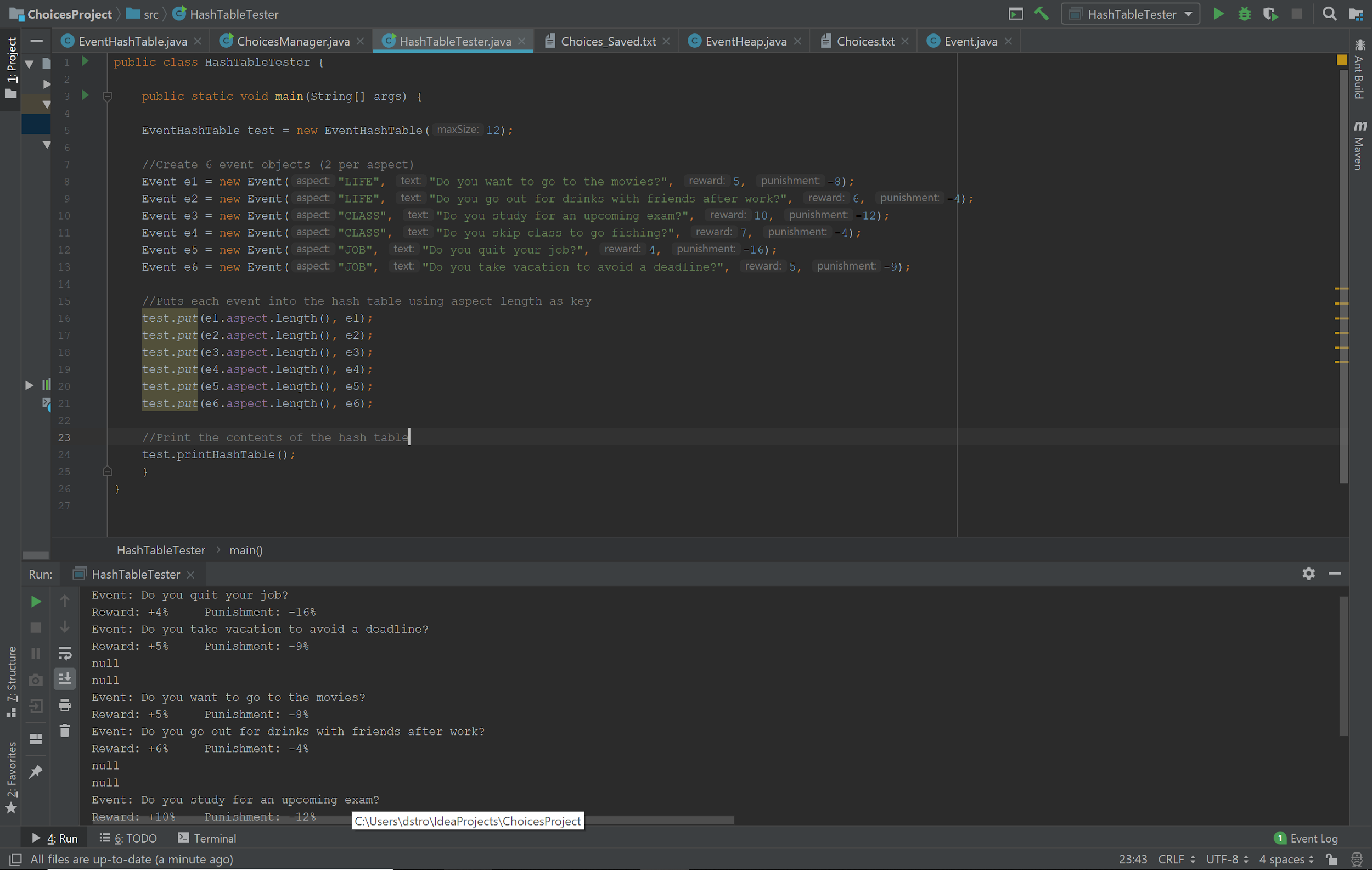
*Example of Warning method*

**Implementation**

We chose to implement the following data structures for this project: An Event Hash Table and an Event Heap. Both of these data structures serve different purposes to the project. The Hash Table is the base data structure that would directly connect with the created text files, while the heap will be the communication between the hash table and the main file.

We used the hash table to serve as the “mass storage” for all of our event objects. The hash table uses a single dimensional array that is subdivided into three subarrays with the following configuration: work related events in the first third of the array, life related events into the second third of the array, and our school events into the last third of the array. The keys for each of these subsections are 3, 4, and 5, for work, life, and school aspects respectively. Because the keys are the differentiating factor between aspects, we expected to encounter a lot of collisions and chose to use a linear probing technique to resolve this issue. While it isn’t the best strategy to practice, we weren’t dealing with a large number of input elements, and the resulting clustering effect from linear probing allowed for easier retrieval for objects later down the road. The last design decision with our hash table was the implementation of our get method. We wanted to ensure the game would be as random as possible, so our get method generates a random number within the range of the subarray and returns a random event object for a desired aspect each time. This technique contrasts heavily with standard practices of a hash table, but it turned out to work best for our design goals.

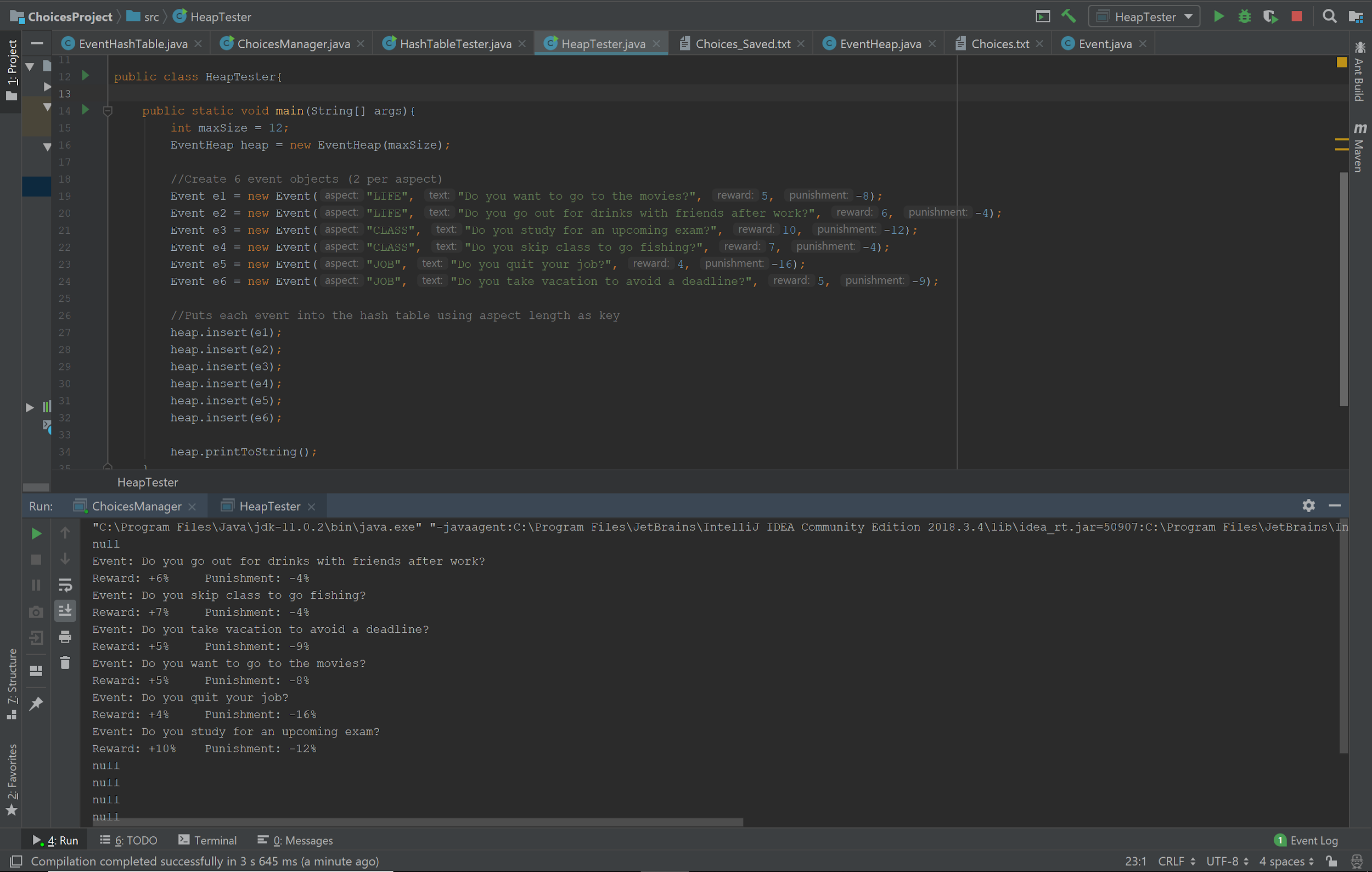
*Example of Hashtable exhibiting clustering*



The heap is used to arrange the data so that it would prioritize the lowest impact value of the event (reward + abs(punishment)) near the top of the heap. This would create increase the difficulty of the game steadily for the player. We implemented a basic heap first, before specializing to accommodate for our needs, by eliminating unnecessary functions, and focus on making sure percolate up and down works based off the impact values of the events.

Expand on heap a bit more…

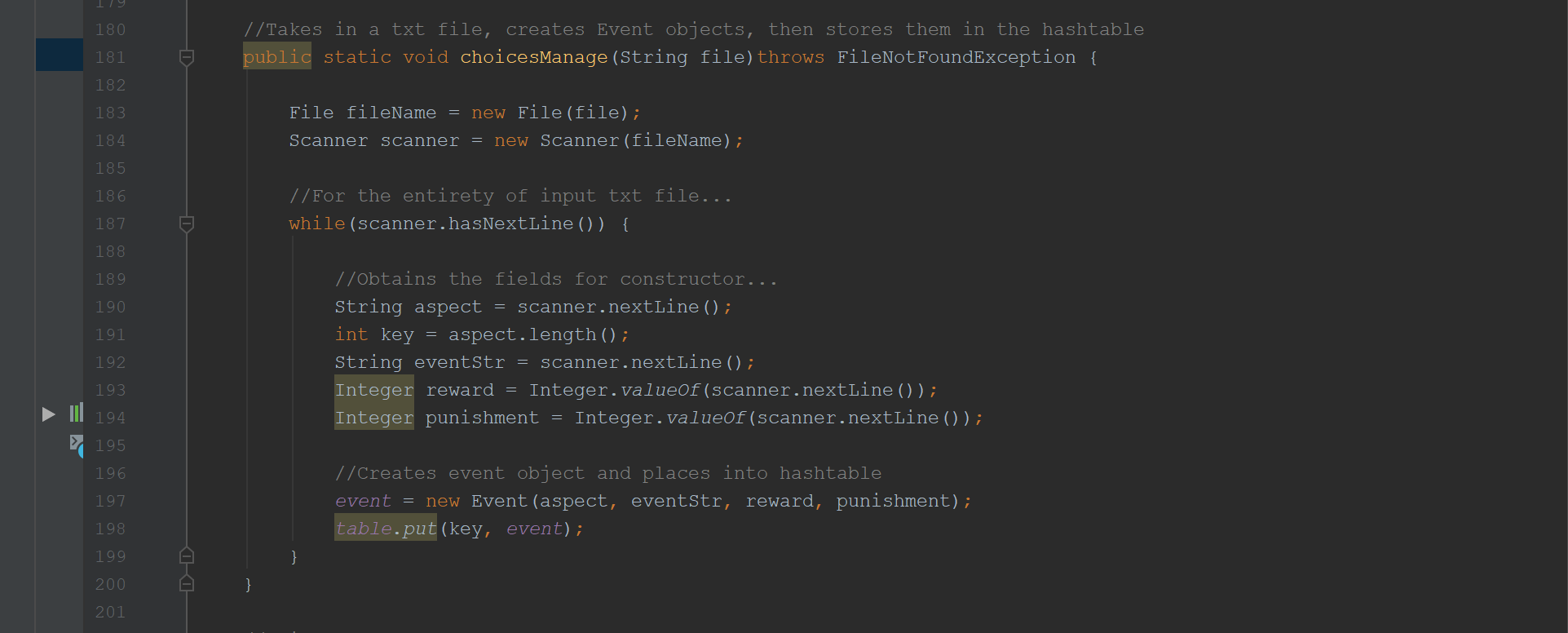
*Example of Heap using array implementation*



**Design Decisions**

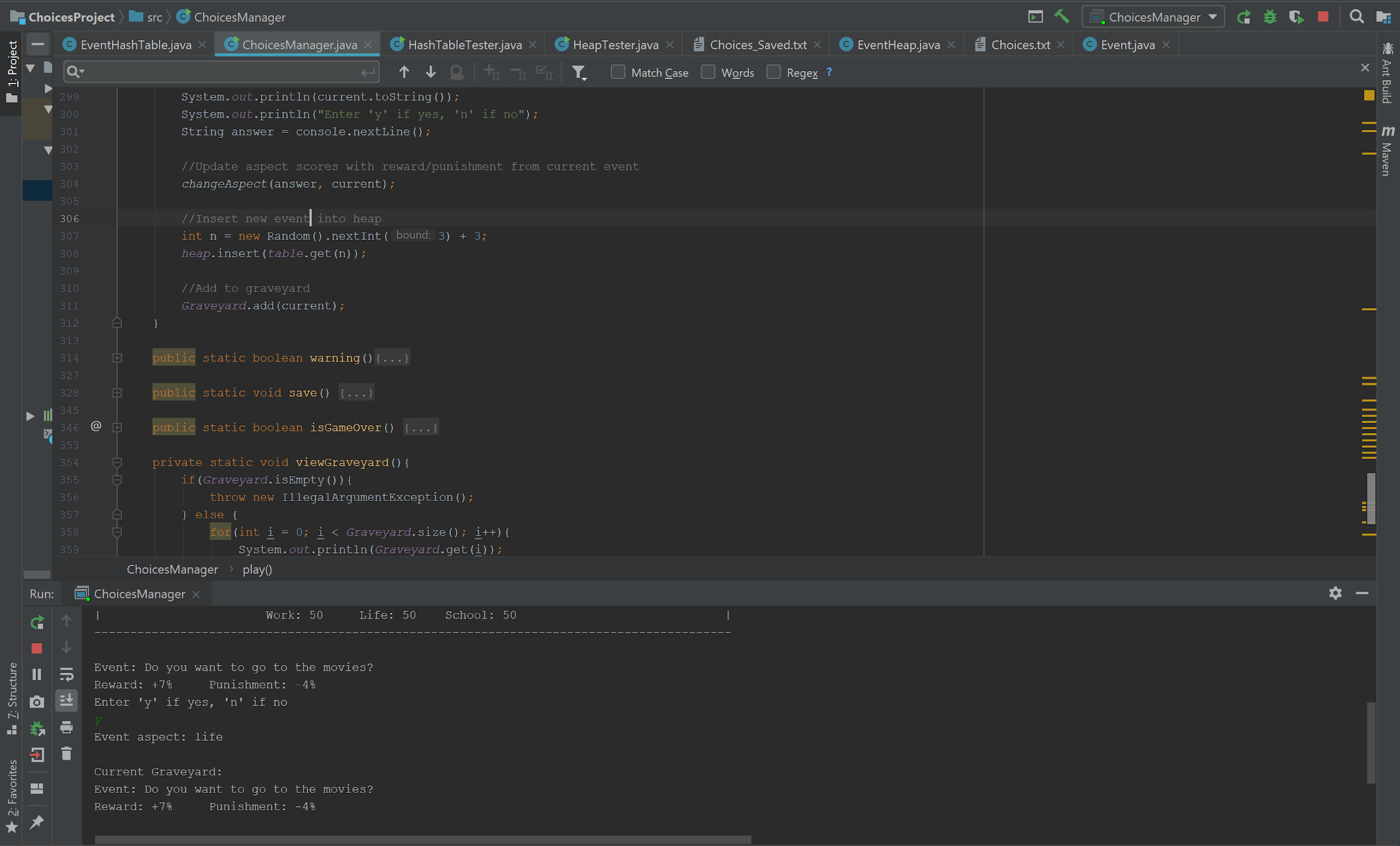
One of our design decisions was to read the data from a text file and create an Event object for each event. The purpose of this stems from the desire to use a hash table to store each object. We knew right away that for each scenario we would have to present the user with a String representing the scenario, and two integers that represent the result of answering yes or no. We decided that storing all the data in an object would allow each event to be unique and easier to group all three of those fields together. This made storing the contents of the input text file in our hash table via our Manager class much easier.

*Example of method to read input from text file*



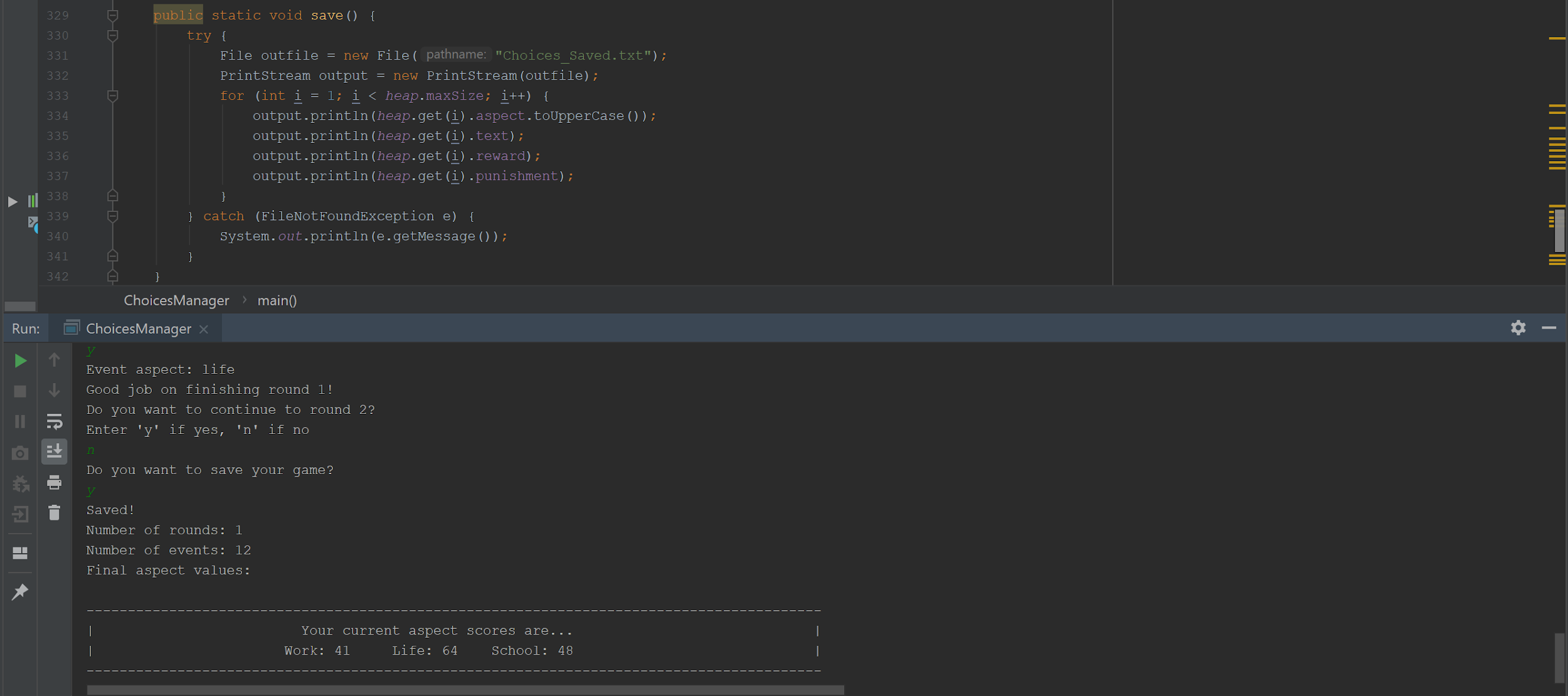
Another big design decision for us was to use a graveyard of some sort to ensure that events wouldn’t be played multiple times in a single round. We initially discussed the idea of creating a second hash table as auxiliary storage, but ultimately decided to use an arrayList for easier implementation. Once an event object is deleted from the Heap, it is placed in a graveyard. As the manager gets a random event from the hash table to insert into the heap, it checks it against the graveyard to ensure that it hasn’t already been played, otherwise it generates a new random event.

*Example of populating the Graveyard after playing an event*

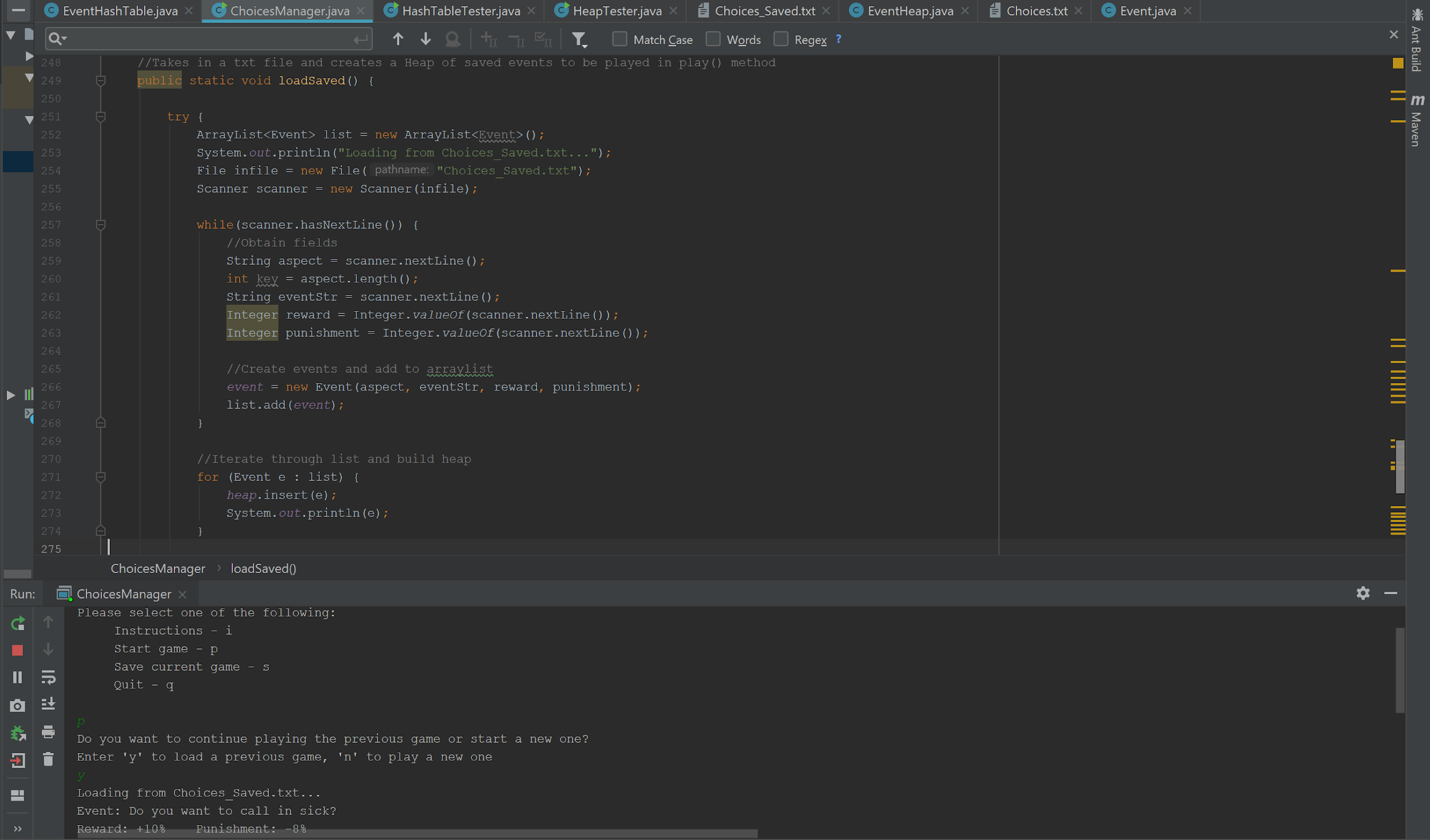


While we didn’t get started working on this until very late in the development of the project, we had always planned to allow the user to save the current state of the game to be reloaded in the future. This proved to be more tricky than it seemed due to everyone’s unfamiliarity working with PrintStream objects. Fortunately for us, there are a plethora of references available and after a few hours of IndexOutOfBounds and NullPointer exceptions resulting from commonly overlooked bugs, we were able to get these features working properly. Before playing a new round, the game asks the user if they would like to start a new game or load a previously saved one. If they choose a new game, they generate the heap from random events as previously stated. When they choose to load a previously saved game, it creates the heap from a second text file and imports the previous aspect scores and resumes where it left off. While this feature wasn’t an absolute necessity for us, it was one that we really wanted to implement.

*Example of the save method and feature*



*Example of the loadSaved method and feature*



Our last design decision comes in the form of one that we weren’t able to achieve. From the early stages, we wanted to give the user the ability to add their own events to the game if they were successful in completing all four rounds. Unfortunately, the implementation of the hash table, heap, and general tying all of these moving pieces together proved to be quite time consuming. Furthermore, this would have also forced us to become familiar with PrintStream objects earlier in the quarter, which although we ended up having to do much later than anticipated might have taken some time away from working out the bugs surrounding the two data structures. Similar to the save and load methods, this wasn’t essential for our program. The inclusion of this would have really allowed us to get one step closer to bringing our initial conceptions of this game to life.

**Testing**

Testing the project has been a more pleasant experience for some than others. I (Don) have spent hours encountering various exceptions and trying to pinpoint exactly why things that shouldn’t be happening happen and vice versa. This is where the benefit of open communication and code review that results from working in a group really proved to be handy. Having Adib and Andrew available to look over my code allowed us to iron out some problems early one. Once we created the hash table, we spent a fair amount of time getting the Manager to play nicely with creating the events and storing them in the proper sections of the array. The get method for this structure also turned out to be quite time consuming primarily with ensuring that the method would not return a null event object (yay recursion!). As for the heap, Adib relied on JGrasps tried-and-true debugger tool and that sped up the debugging process exponentially! Once both data structures were implemented, we focused on ensuring that we could play a single round of six questions (this would ultimately increase in the final product). Keep adding to this…

**Goals / Future Work**

For the most part, we achieved everything we originally set out to accomplish – with the exception of the GUI. We wanted to create a random game that was designed to “help” the user along the way. Unfortunately, there are some pieces that weren’t able to be completed primarily due to lack of time and unavailability of certain members. One of our primary goals was to present a finished and polished product. This relied heavily on a GUI to ensure that users wouldn’t have to squint their eyes while reading and interacting with the console. Unfortunately, that task never got accomplished and we ultimately had to scrap it. Provided more time, a GUI would definitely be our next priority. Similarly, we wanted to incorporate a “hint” feature for the user that would direct them to the answer for a given scenario that would be most beneficial for them. I believe that this could have easily been achieved if we had just a few more days to continue working on the program. Similar to the hint feature, we wanted to allow the client the opportunity to contribute to the games success by creating their own events, provided they completed the game successfully first. This was inspired by the CS145 “20 Questions” lab. Unfortunately this feature wasn’t high on our priority list and falls into the category of “future projects.”

**Contributions**

Teamwork and open communication was paramount throughout this project. In the beginning, Don tackled the Event object and a significant portion of the hash table. Input from everyone was essential in the planning stages for how we wanted our hash table to work. After a week trial and error, we finally had a working product. Next, Adib tackled the heap in a similar fashion using the input provided from the group collectively. Andrew handled a lot of the design aspects for the game itself, and played an instrumental role in tying the different classes together in the manager. The manager was a collective effort, with everyone testing for bugs and adding features as needed. Lastly, Micah took on the project of creating a GUI for the game, although due to the complexity of implementing a GUI and the lower priority in terms of things that *needed* to get done, we weren’t able to achieve this goal.

**Final Thoughts**

This was an exciting project that each one of us were eager to work on. From the beginning, group discussions fueled our development efforts. One of the most intriguing aspects for our group was the vastly different directions that everyone envisioned for our project. This level of creativity inspired the creation of the load/save methods and the randomization aspect of our hash table. This culture embraced creativity and allowed each author to have a part in the creation of this game. Overall, we are very pleased with the outcome of our project. While we sure would have enjoyed more time to implement a few extra features, we understand that time on a quarter-system moves quite quickly. Nevertheless, we achieved the goals that we set out to accomplish and created a game that each one of us thoroughly enjoys.