To start the game, run the Driver.java class. Use the terminal to interact with the program, a window will appear that will display your name and current stats.

IRL Adventures is an application that gamifies the everyday life of a college student. The user creates a character and decides on how that character spends their time. However, just like normal life, seemingly random events can happen throughout a person’s day, affecting the player's productivity for the day. The time in each day is limited, while the number of days to play through is dictated by the user.

The actual character itself with their name, level, stats, and time left in the day is stored in the Character class which implements the Singleton design pattern. This pattern is extremely useful in this scenario given that only one character may be used at a time. The fact that Character is a Singleton means that anytime the character’s stats are retrieved or updated, we know that it is coming from the same object.

The Memento pattern is used to store and retrieve the user’s character data in-between sessions. Nested within the Character class, Memento creates an object that stores the character’s data such as name, level, and stats. The Caretaker class stores the last save file, and also writes and retrieves this data to and from a text file so that the data is stored in-between sessions.

The user’s character starts off a day of the relaxed state. The PlayerDay class is the character’s current day but the way it flows depends on the state it is run in. The State pattern allows the user to go through the day similarly to the other states but with small changes depending on the player’s mood. When random events occur through the player’s productive day, their mood might change.

The Observer pattern allows the amount of random events to be recorded, then changes the state according to how many occurred. The observer gets notified by each random event, keeping track and making changes to the flow.

Throughout each day the user has the option to pick the activity they want to do. They operate all similarly, but with a key difference being the xp getting awards to the correct player statistics. The use of the Template pattern is used here ensure the process of going through the activity is the same, but the difference in each is implemented differently according to the activity. The template pattern also receives the state which the activity is called and adjusts the character’s xp amount according to the state/mood of the player

As the player goes about their day, random events get triggered which issues a skill check. A scenario gets displayed to the user and a skill check is issued that determines if the user’s character has high enough stats to pass. If the character passes the skill check, they level up. As the player gains higher levels, more difficult skill checks need to be issued. Here we used the Abstract Factory pattern to issue skill checks dynamically based off of the user’s level.

Abstract Factory:

This was used to make the sub windows for the GUI part of the program, I didn’t know what would be needed for the different subwindows but they used the same implementation so it was a good thing to use. It also makes it easy to make new types of subwindows and groups them together for any similar use in the future.

Composite:

This was used on the buttons and the panels in the character screen of the first GUI. I used this so when the character got enough data to level up we would be able to change what was being said on the character screen. I gave them all the update methods using a common interface between them. Now when the character called the level up method they would just need to pass that on to the window and it would all change. They specifically only need to pass it on to the MyPanel which would then do the same work for the rest of the buttons. It also allows us to add more methods that might affect all the buttons like a change color method. Or if there is something that I need to add to that whole update method that I forgot, like a JLabel, it is easy to implement that for any GUI component.

ModelViewController:

This pattern was actually implemented without knowing the name for it at first, I use a button listener to listen to the window so it can see if a button was clicked. When that button is clicked I then have the button listener look at what was clicked and decide what type of subwindow to open up and how that sub window should look. ButtonListner is the Controller, Characterscreen is the view and the subwindows and update methods which are called through the button listener are the model. I just had to fully remove the buttonlistener class and make a few improvements. This is an incredibly helpful tool because it makes the code easy to read and improves the reusability.