Test Driven Development prompts us to first write the test for a code that is to be implemented and then write the actual code implementation. We will follow this development method to write our code for the Chandrayan movement.

Note: The testing is done in JavaScript using jest.

Firstly we define a function. Say, **navigateChandrayan** is the function that will be used for navigating the Chandrayan.

1. **Writing navigateChandrayan arguments:**
   navigateChandrayan will take three arguments, initialPosition, initialDirection and commands.

   Now, we need initialPosition to be an object having three attributes, x, y and z that must be of type number.

   Another thing to keep in mind is that interspace communication is costly(in terms of time and money) so it becomes necessary to check that the information being sent is correct and relevant. We need to make sure that we send Some commands to the Chandrayan and not an empty list.

   We make tests for these arguments and when the test fails, we write necessary code to pass them.

```
function testNavigateSpacecraft() {
  //
}

describe("Tests for navigateChandrayan", () => {
  test("Testing arguments", () => {
    expect(typeof initialPosition.x).toBe(Number);
    expect(typeof initialPosition.y).toBe(Number);
    expect(typeof initialPosition.z).toBe(Number);
    expect(commands).not.toBe([]);
  });
});
```

But the code fails as we have not yet defined initailPosition and commands. Let's define them and run the test.

```
describe("Tests for navigateChandrayan", () => {
  test("Testing initial position", () => {
    const initialPosition = { x: 0, y: "x", z: 0 };

    expect(typeof initialPosition.x).toBe("number");
    expect(typeof initialPosition.y).toBe("number");
    expect(typeof initialPosition.z).toBe("number");
  });
  test("Test if command is Empty", () => {
    const commands = [];
    const empty = [];
    expect(commands).not.toEqual(empty);
  });
});
```

Again the tests fail because we have given incorrect values(But that means we're on the right track!).

We provide valid values

We passed the test.

2. **Creating directions array:**
   Now that we have validated our arguments, next we need to work on the logic of the code.First focusing on the directions.

   But, the directions that we know are relative. That means we understand the directions with reference to something. But in space there is no reference similar to that on Earth.
   For example, on Earth we know that the Sun rises in the East, but in space there are Millions of Suns! So where is East?

   To tackle this problem, we will define the directions using Cartesian Coordinates. But, how will we know where we will reach if we go Left from Northor down from West?

```
describe("Tests for directions in space", () => {
  test("Testing directions", () => {
    expect(directions["N"]["l"]).toBe("W");
  });
});
```

Our test fails as there is no way to know what will be on the left, right, up or down from a direction.
Let’s define this using an object named directions.

```
describe("Tests for directions in space", () => {
  test("Testing directions", () => {
    const directions = {
      N: { l: "W", r: "E", u: "U", d: "D" },
      E: { l: "N", r: "S", u: "U", d: "D" },
      W: { l: "S", r: "N", u: "U", d: "D" },
      S: { l: "E", r: "W", u: "U", d: "D" },
      U: { l: "W", r: "E", u: "S", d: "N" },
      D: { l: "W", r: "E", u: "N", d: "S" },
    };
    expect(directions["N"]["l"]).toBe("W");
  });
});
```

We passed the test.

3. **Executing commands**

   Now that we have directions and commands, let's run them!

   But, first we need to check if the commands are valid or not. What if the given command does not exist in the pre-defined commands? We need to check this.

```
describe("Test for executing commands", () => {
  test("Test if command is Empty", () => {
    const commands = ["f", "r", "u", "b", "x"];
    const preDefCommands = ["f", "r", "u", "b", "l"];
    let failed = false;
    for (command of commands) {
      if (preDefCommands.indexOf(command) == -1) {
        failed = true;
        break;
      }
    }
    expect(failed).toBe(false);
  });
});
```

   Here, we declare an array that stores all the predefined commands and loop through the array and check if the command exists. If not, the test fails.

   After testing the commands now we need to execute them.
   We need to do two things:

   a. Move the Chandrayan forward or backward.
   b. Rotate the Chandrayan to change its direction.

   We will do this using **move** and **rotate** functions

4. Define move and rotate functions:

Rotate:
Rotate function is simple, if we encounter a ‘l’, ‘r’, ‘u’ or ‘d’ command, we rotate the Chandrayan.
How? Using the directions array we defined earlier!

Let’s test it

```
function rotate(currentDirection, command) {
  const directions = {
    N: { l: "W", r: "E", u: "U", d: "D" },
    E: { l: "N", r: "S", u: "U", d: "D" },
    W: { l: "S", r: "N", u: "U", d: "D" },
    S: { l: "E", r: "W", u: "U", d: "D" },
    U: { l: "W", r: "E", u: "S", d: "N" },
    D: { l: "W", r: "E", u: "N", d: "S" },
  };
  return directions[currentDirection][command];
}
```

```
describe("Test for rotate and move functions", () => {
  test("Test rotate functions", () => {
    const currentDirection = "N";
    expect(rotate(currentDirection, "l")).toBe("W");
  });
});
```

Similarly, we write move function:

```
function move(currentPosition, currentDirection, val) {
  if (
    current DirectionS" ||
    currentDirection == "W" ||
    currentDirection == "D"
  ) {
    val *= -1;
  }
```

```
  if (currentDirection == "N" || currentDirection == "S") {
    currentPosition.y += val;
  } else if (currentDirection == "E" || currentDirection == "W") {
    currentPosition.x += val;
  } else {
    currentPosition.z += val;
  }
  return currentPosition;
}
```

Finally, we take all these parts of the code and put it together to obtain our code that will help Chandrayan navigate through the space!!!