

Reflections on "Clean Code" (kapitel 2-11)

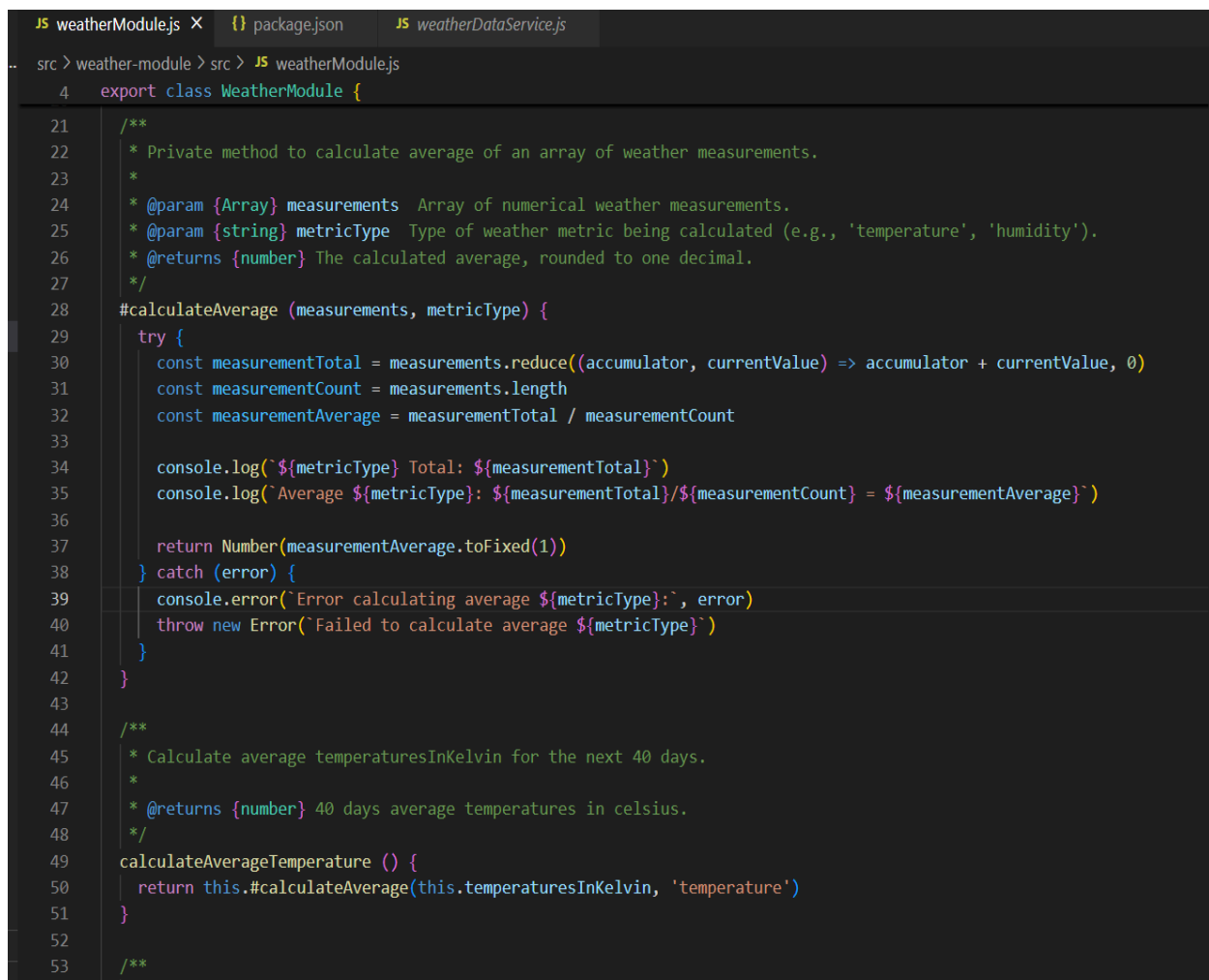
1. Meaning Names (Chapter 2)

See the reflection at the weather module repository.

2. Functions (Chapter 3)

See the reflection at the weather module repository.

According to the "Don't Repeat Yourself" (DRY) principle, I updated the code in the weather module of my weather application. Here I refactored it to use a single private function to calculate the average value and this avoids redundancy and adheres to the DRY principle:



```
JS weatherModule.js X {} package.json JS weatherDataService.js
src > weather-module > src > JS weatherModule.js
4 export class WeatherModule {
21 /**
22  * Private method to calculate average of an array of weather measurements.
23  *
24  * @param {Array} measurements Array of numerical weather measurements.
25  * @param {string} metricType Type of weather metric being calculated (e.g., 'temperature', 'humidity').
26  * @returns {number} The calculated average, rounded to one decimal.
27  */
28 #calculateAverage (measurements, metricType) {
29   try {
30     const measurementTotal = measurements.reduce((accumulator, currentValue) => accumulator + currentValue, 0)
31     const measurementCount = measurements.length
32     const measurementAverage = measurementTotal / measurementCount
33
34     console.log(`${metricType} Total: ${measurementTotal}`)
35     console.log(`Average ${metricType}: ${measurementTotal}/${measurementCount} = ${measurementAverage}`)
36
37     return Number(measurementAverage.toFixed(1))
38   } catch (error) {
39     console.error(`Error calculating average ${metricType}:`, error)
40     throw new Error(`Failed to calculate average ${metricType}`)
41   }
42 }
43
44 /**
45  * Calculate average temperaturesInKelvin for the next 40 days.
46  *
47  * @returns {number} 40 days average temperatures in celsius.
48  */
49 calculateAverageTemperature () {
50   return this.#calculateAverage(this.temperaturesInKelvin, 'temperature')
51 }
52
53 /**
```

Figure 1: Have only one private function to calculate average value

3. Comments (Chapter 4)

I noticed that some comments in my code were redundant, such as those added only to satisfy JSDoc requirements. These "noise comments" made the code harder to read because the function names were already self-explanatory.

```
/**
 * Test average humidity calculation.
 */
const testAverageHumidityCalculation = () => {
  const expected = 5.4
  const actual = weatherModule.countAverageHumidity()

  if (actual === expected) {
    console.log('✅ TC2 - Humidity Calculation: PASSED')
    console.log(` Expected: ${expected}%`)
    console.log(` Actual: ${actual}%`)
  } else {
    console.log('❌ TC2 - Humidity Calculation: FAILED')
    console.log(` Expected: ${expected}%`)
    console.log(` Actual: ${actual}%`)
  }
}

/**
 * Test wind speed calculation.
 */
const testWindSpeedCalculation = () => {
  const expected = 2.8
  const actual = weatherModule.countAverageWindSpeed()

  if (actual === expected) {
    console.log('✅ TC3 - Wind Speed Calculation: PASSED')
    console.log(` Expected: ${expected}m/s`)
    console.log(` Actual: ${actual}m/s`)
  } else {
    console.log('❌ TC3 - Wind Speed Calculation: FAILED')
    console.log(` Expected: ${expected}m/s`)
    console.log(` Actual: ${actual}m/s`)
  }
}
```

Figure 2: Show the unnecessary JSDoc

I have redundant comments such as `console.log()` during my applications development, because i want to see at the console which value i get back, but after that those should be removed when the application and make the code cleaner:

```

const rainfall = weatherDataList.map(item => item.rain && item.rain['3h'] ? item.rain['3h'] : 0)
// Check if item.rain exists and has a property '3h'
console.log('rainfall form app.js console.log:' + rainfall)

const weatherModule = new WeatherModule(temperaturesInKelvin, humidities, windSpeeds, rainfall)

const averageTemperature = await weatherModule.calculateAverageTemperature()
// console.log('The average temperature in kelvin for the next 40 days is around: ' + averageTemperature + 'K.')

const averageTemperatureInCelsius = await weatherModule.convertKelvinToCelsius(averageTemperature)
console.log('The average temperature in kelvin is around ' + averageTemperature + ' K' + ' which is around (' + averageTemperature + ' - 2
// console.log('The average temperature in kelvin is ' + averageTemperature + 'K' + ' which is around ' + averageTemperatureInCelsius + '°

const averageHumidity = await weatherModule.calculateAverageHumidity()
// console.log('The average humidity for the next 40 days is around: ' + averageHumidity + '%.')

const averageWindSpeed = await weatherModule.calculateAverageWindSpeed()
// console.log('The average wind speed for the next 40 days is around: ' + averageWindSpeed + 'm/s.')

const maxRainfall = await weatherModule.calculateMaximumRainfall(rainfall)
// console.log('The maximum rainfall for the next 40 days is around: ' + maxRainfall + 'mm.')

```

Figure 3: Show the console.log() comment line

After i removed console.log(), The code is condensed to 6 lines:

```

const weatherModule = new WeatherModule(temperaturesInKelvin, humidities, windSpeeds, rainfall)
const averageTemperature = await weatherModule.calculateAverageTemperature()
const averageTemperatureInCelsius = await weatherModule.convertKelvinToCelsius([averageTemperature])
const averageHumidity = await weatherModule.calculateAverageHumidity()
const averageWindSpeed = await weatherModule.calculateAverageWindSpeed()
const maxRainfall = await weatherModule.calculateMaximumRainfall(rainfall)

```

Figure 4: Show the after removed console.log() comment line

But some commandant is necessary to keep as a student, such as the inspiration where I get from internet:

```

convertKelvinToCelsius (inputTemperatureKelvin) {
  try {
    return Number(inputTemperatureKelvin - 273.15).toFixed(1)
    // Inspiration: https://www.metric-conversions.org/temperature/kelvin-to-celsius.htm
  } catch (error) {
    console.error('Error converting Kelvin to Celsius:', error)
    throw new Error('Failed to convert temperature from Kelvin to Celsius')
  }
}

```

Figure 5: Show the necessary comment line

4. Formatting (Chapter 5)

1. Vertical density implies close association. From figure 3 to figure 4 above, I learned the importance of keeping related code together. Removing redundant console.log() statements improved the vertical density of my code, making associations between lines clearer.

2. I agree with the author about horizontal formatting, that we should keep our lines short (80 - 100 lines) to improve readability. Lines longer than 100 characters decrease readability. I resolved this by breaking them into multiple lines:

```
32
33     const url = `${this.baseUrl}/geo/1.0/direct?q=${encodeURIComponent(city)},${encodeURIComponent(country)}&limit=18
34     const response = await fetch(url)
35     console.log(url)
36
37     if (!response.ok) {
38         throw new Error(`Failed to fetch coordinates: Status code ${response.status}`)
39     }
40     const data = await response.json()
41
42     if (data.length === 0) {
43         console.log('No data returned', data)
44         throw new Error('City not found or invalid country code')
45     }
46
47     const matchedLocation = data.find(location => location.name.toLowerCase() === city.toLowerCase() && location.cour
48
49     if (!matchedLocation) {
50         throw new Error('No matching location found for ${city}, ${country}')
51     }
```

Figure 6: Line 33 and 47 have too many characters in one line

Solved this problem braking it into multiple lines:

```
32
33     const url = `${this.baseUrl}/geo/1.0/direct?q=${encodeURIComponent(city)},
34     ${encodeURIComponent(country)}&limit=18appid=${this.apiKey}`
35
36     const response = await fetch(url)
37     console.log(url)
38
39     if (!response.ok) {
40         throw new Error(`Failed to fetch coordinates: Status code ${response.status}`)
41     }
42     const data = await response.json()
43
44     if (data.length === 0) {
45         console.log('No data returned', data)
46         throw new Error('City not found or invalid country code')
47     }
48
49     const matchedLocation = data.find(
50         location => location.name.toLowerCase() === city.toLowerCase() &&
51         location.country.toLowerCase() === country.toLowerCase()
52     )
53
```

Figure 7: Line 33 and 49 show after lines breaks

5. Objects and Data Structures (Chapter 6)

In my application, I use JavaScript's class syntax, which includes features like private methods (e.g., #calculateAverage in 'WeatherModule') to encapsulate functionality. This private method calculates the average of weather measurements, such as temperature or humidity, without exposing its implementation to the rest of the application. This approach safeguards internal logic and reduces the risk of external attacks. And in the future I consider using TypeScript to enforce better type safety.

6. Error Handling (Chapter 7)

I mostly use try-catch blocks for error handling in my application. For example, when fetching data from an API, it's critical to catch errors and specific error messages that help with debugging to ensure the application doesn't crash. And I am wondering if it is necessary to create custom error classes for different types of errors (At small applications)?

```
try {
  setError('');
  const moduleResult = await fetchAndCalculateWeatherData(city, country);
  setWeatherData({
    temperature: { ...
  },
  humidity: { ...
  },
  windSpeed: { ...
  },
  rainfall: { ...
  }
});
} catch (error) {
  console.error(error);
  setError("Sorry, we couldn't fetch the weather data. Please try again later.");
}
```

Figure 8: Try-catch at App.js

7. Boundaries (Chapter 8)

I used an OpenWeatherMap API to fetch weather data. As the author suggests, it is crucial to understand the boundaries of external libraries or APIs. But even if I understand the documentation, I will find uncontrollable errors. Initially, when I was looking at the documentation of the API, I thought it provided a 5-day weather forecast. However, I found that when I called this API it gave me a 40-day forecast with an interval of 3 hours. So I modified my application based on the weather data I got from OpenWeatherMap.

8. Unit Tests (Chapter 9)

1. I agree with the author 'Test code is just as important as production code', I use unit tests to validate the main functions of my application, ensuring it works as expected. My application will not complete if any one of them is missed or test failure.
2. I agree that readability makes a clean test. Readable and concise tests are critical for understanding failures. I ensure function names and test data are simple and clear.

Readable tests save time during debugging and enhance collaboration with team members.

```
const runWeatherModuleTests = () => {
  const testData = {
    temperaturesInKelvin: [173.15, 273.15, 400.15, 200],
    humidities: [4, 6, 2, 8, 7],
    windSpeeds: [1, 5, 2, 3],
    rainfall: [2, 5, 7, 9, 2, 4]
  }

  > const weatherModule = new WeatherModule(...)
  >
  > /** ...
  > const testAverageTemperatureCalculation = () => { ...
  > }
  >
  > /** ...
  > const testAverageHumidityCalculation = () => { ...
  > }
  >
  > /** ...
  > const testWindSpeedCalculation = () => { ...
  > }
  >
  > /** ...
  > const testMaxRainfallCalculation = () => { ...
  > }
```

Figure 9: Shows unit tests

9. Classes (Chapter 10)

I agree with the principle that classes should be small and focused. If a class takes on too much responsibility, it becomes difficult to maintain and debug. In my application, I refactored the App.js file to separate responsibilities into smaller classes, like ‘WeatherDataService’ and ‘WeatherModule’. This separation makes the application more scalable and easier to debug.

10. Systems (Chapter 11)

In my application, I followed the principle of separation of concerns by using multiple classes, such as ‘WeatherDataService’ and ‘WeatherModule’, to handle distinct responsibilities. This approach makes it easier to modify or replace one part of the system without affecting others. By isolating specific functionalities into dedicated components, the design enhances maintainability and scalability.

The ‘fetchAndCalculateWeatherData’ function indirectly implements dependency injection internally by creating instances of ‘WeatherDataService’ and ‘WeatherModule’. While this approach works well, explicitly passing these instances as arguments could further improve testability (e.g., by allowing class mocking) and overall flexibility.