CS 731: Software Testing November 27, 2023

Software Testing Project

Instructor: Prof. Meenakshi D'souza

Kaushik Mishra (IMT2020137)
Srinivas Manda(IMT2020001)

Introduction and Project Overview

We implemented a **Server-side web application testing method using the CIM (Component Interaction Model) and ATG(Application Transition Graph) paradigm** on the ReactJS game "Wordle" implemented by us in a project elective.

Wordle, a word-guessing game which challenges players to decipher a secret five-letter word within six attempts. With each guess, the game provides feedback, indicating correct letters in their positions with specific colors. Players strategically narrow down possibilities by analyzing feedback and refining subsequent guesses.

The GitHub Repository of our project is linked here Wordle.

Please run the following commands to compile and run the testcases.

```
npm install
npm test
```

If there are any Critical Vulnerabilities then please run the following command until there aren't any,

```
npm audit fix
```

Project Setting and Designed Test Cases

0.1 Component Interaction Model :

We model the individual components of the game in the form of their atomic components and test them accordingly.

1. **Keyboard.tsx :** We first test the Virtual keyboard component of the project ensuring that atomic sections within it work correctly, and all of them combine achieved their intended functionality. We first identify the atomic sections within the code as,

   (a) **The Virtual keyboard is rendered correctly onto the WebApp :**

```
it('CIM: renders the Keyboard component correctly', () => {
    const { container } = render(<Keyboard {...defaultProps} />)
    const text = screen.getByText('Q')
    expect(text).toBeInTheDocument()
    const text1 = screen.getByText('Enter')
    expect(text1).toBeInTheDocument()
    // Add more assertions as needed
})

```

(b) **OnClick of any Character, OnEnter, OnDelete functionalities work as intended by a Virtual Keyboard :**

```
it('CIM: fires onClick event for a key', () => {
    const { getByText } = render(<Keyboard {...defaultProps} />)
    const keyElement = getByText('A') // Replace 'A' with any key you want to test
    fireEvent.click(keyElement)
    expect(onCharMock).toHaveBeenCalledWith('A')
})

it('CIM: fires onDelete and onEnter events on keyboard events', () => {
    const { container } = render(<Keyboard {...defaultProps} />)
    fireEvent.keyUp(container, { key: 'Enter', code: 'Enter' })
    expect(onEnterMock).toHaveBeenCalledTimes(1)

    fireEvent.keyUp(container, { key: 'Backspace', code: 'Backspace' })
    expect(onDeleteMock).toHaveBeenCalledTimes(1)

    // Add more keyboard event simulations and assertions as needed
})

```

2. **App.tsx :** This component contains the code for the main body and word validation functionality of the entire project. We identify the atomic sections within the code and test them accordingly,

   (a) **Checking the presence or absence of guessed letters :**

```
describe('getGuessStatuses', () => {
  test('CIM: guess statuses', () => {
    expect(getGuessStatuses('ABCDE', 'EDCBA')).toEqual([
      'present',
      'present',
      'correct',
      'present',
      'present',
    ])
    expect(getGuessStatuses('ABCDE', 'VWXYZ')).toEqual([
      'absent',
      'absent',
      'absent',
      'absent',
      'absent',
    ])
    expect(getGuessStatuses('ABCDE', 'ABCDE')).toEqual([
      'correct',
      'correct',
      'correct',
      'correct',
      'correct',
    ])

    expect(getGuessStatuses('BOSSY', 'SASSY')).toEqual([
      'absent',
      'absent',
      'correct',
      'correct',
      'correct',
    ])
  })
})

```

(b) **Checking whether the appropriate status is assigned to the correctly guessed letter :**

```
test('CIM: renders Cell component correctly with correct status', () => {
  const { container } = render(<Cell {...cellProps} status="correct" />)

  const mainDiv = container.querySelector('div')
  expect(mainDiv).toHaveClass('correct')
})

```

(c) Many other unit test are done within the file to check whether everything is rendered, guesses are validated correctly, colors of the guessed letters are shown appropriately, and many more.

0.2 Application Transition Graph :

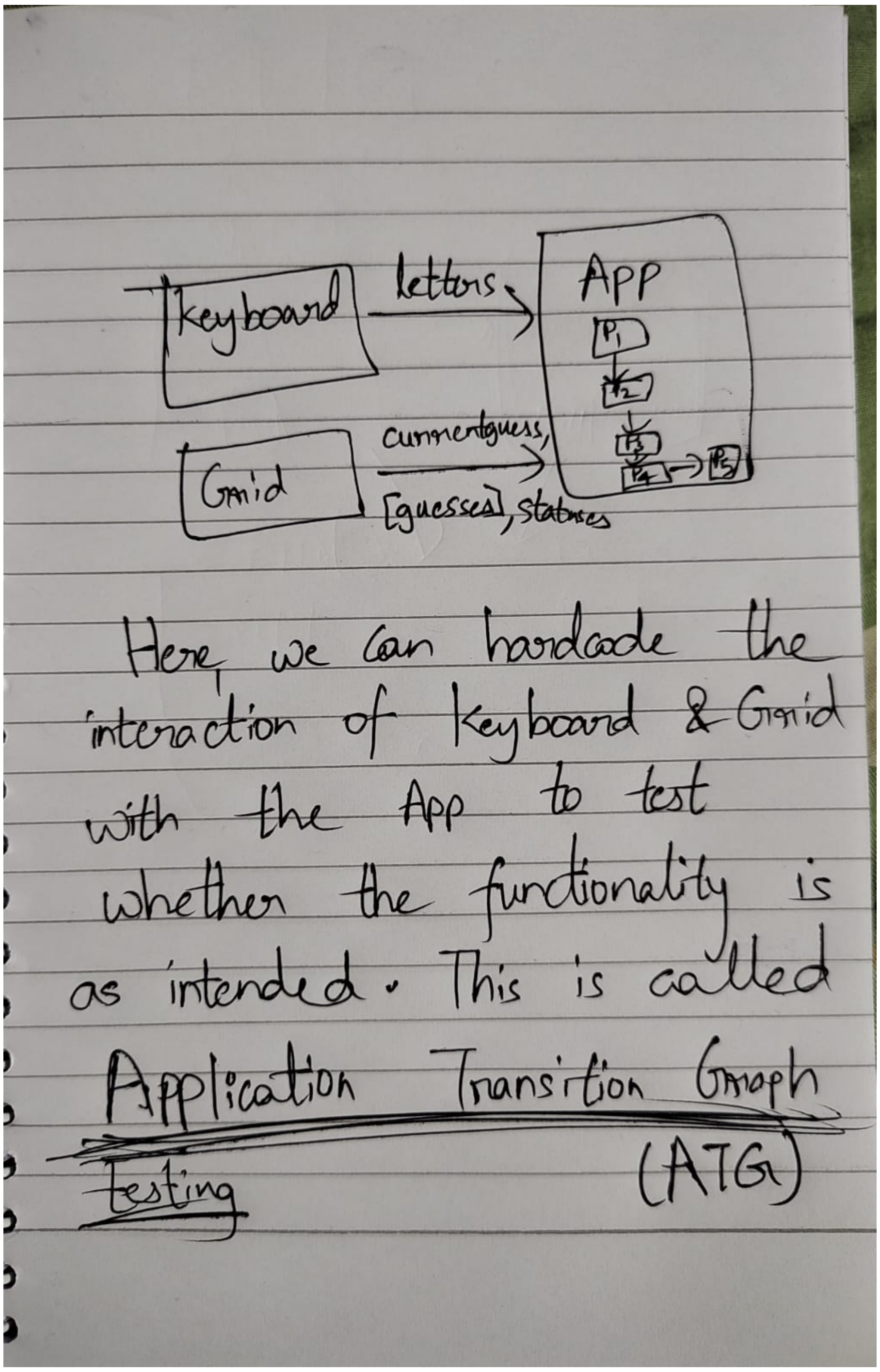
We test the interaction of components and doing integration testing of sorts using graphs to analyse the interaction of components,

1. **App.tsx :** This component contains the code for the main body and word validation functionality of the entire project. We hardcode the output of function calls of some external interlinked components to test this code for checking if it is working as intended,

   (a) **We test the functionality of the app by simulating the function of the virtual keyboard by hardcoding it :**

```
test('Integration: Clicking a key on the onscreen keyboard', () => {
  const { getByText, getByLabelText, getByTestId, container } = render(<App />)
  // console.log(container)
  const GridDiv = container.querySelector('div')
  Simulate.click(getByText('Z'))
  const t = screen.getAllByText('Z')
  for (let i = 0; i < t.length; i++) {
    expect(t[i]).toBeInTheDocument()
  }
})

```


Keyboard
letters
App
P1
P2
P3
P4
P5
Grid
currentguess,
[guesses], statuses

Here, we can hardcode the interaction of Keyboard & Grid with the App to test whether the functionality is as intended. This is called Application Transition Graph Testing (ATG)

Figure 1: The Application Transition Graph (ATG) based testing of the App.tsx file.

(b) **Testing the validation of inputs from the on screen keyboard to the grid cells.**

```
test('Integration: Clicking all 5 keys on the onscreen keyboard and submitting', ()
    => {
  const { getByText, getByLabelText, getByTestId, container } = render(<App />)
  const GridDiv = container.querySelector('div')
  // console.log(container)
  const lst = container.children
  for (let i = 0; i < lst.length; i++) {
    // console.log(lst[i].className)
  }

  // console.log(screen.getAllByRole('cell').length)

  const it1 = screen.getAllByText('S')
  const it2 = screen.getAllByText('H')
  const it3 = screen.getAllByText('A')
  const it4 = screen.getAllByText('R')
  const it5 = screen.getAllByText('D')
  Simulate.click(getByText('S'))
  Simulate.click(getByText('H'))
  Simulate.click(getByText('A'))
  Simulate.click(getByText('R'))
  Simulate.click(getByText('D'))
  const t1 = screen.getAllByText('S')
  const t2 = screen.getAllByText('H')
  const t3 = screen.getAllByText('A')
  const t4 = screen.getAllByText('R')
  const t5 = screen.getAllByText('D')
  let f1
  let f2
  let f3
  let f4
  let f5
  // console.log(t1[0])
  for (let i = 0; i < t1.length; i++) {
    // console.log(t1[i])
    expect(t1[i]).toBeInTheDocument()
  }
  for (let i = 0; i < t2.length; i++) {
    expect(t2[i]).toBeInTheDocument()
  }
  for (let i = 0; i < t3.length; i++) {
    expect(t3[i]).toBeInTheDocument()
  }
  for (let i = 0; i < t4.length; i++) {
    expect(t4[i]).toBeInTheDocument()
  }
  for (let i = 0; i < t5.length; i++) {
    expect(t5[i]).toBeInTheDocument()
  }
  expect(t1.length - it1.length).toBe(1)
  expect(t2.length - it2.length).toBe(1)
  expect(t3.length - it3.length).toBe(1)
  expect(t4.length - it4.length).toBe(1)
  expect(t5.length - it5.length).toBe(1)

  Simulate.click(getByText('Enter'))

  const cells = screen.getAllByRole('cell')
  expect(cells[0]).toHaveClass('present')
  expect(cells[1]).toHaveClass('absent')
  expect(cells[2]).toHaveClass('present')
  expect(cells[3]).toHaveClass('absent')
  expect(cells[4]).toHaveClass('absent')
})

```

Diagrams

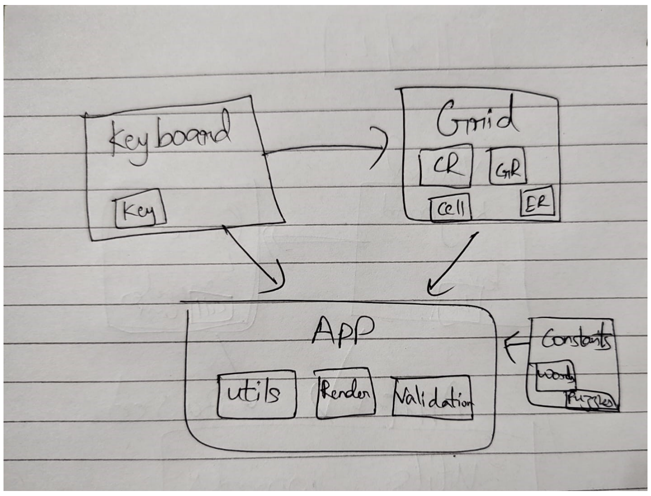


Figure 2: The Component level interaction of the entire project

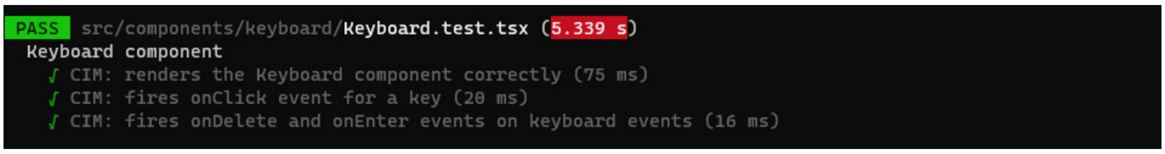
TechStack used

The Project uses ReactJS written in Typescript (A superscript of JavaScript) and testing is done in the same. We use the React Testing Library, and Jest for testing the project

Results

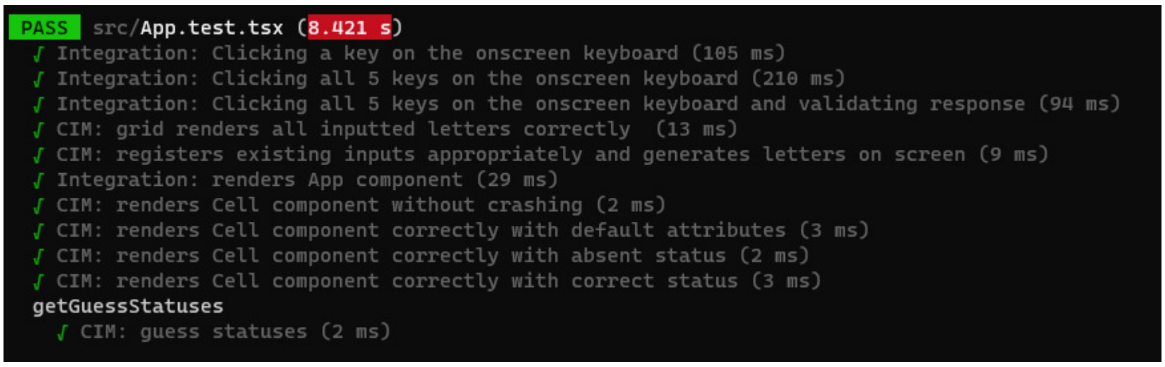
We run the following command to run all of the tests within the project.

```
npm test
```



```
PASS  src/components/keyboard/Keyboard.test.tsx (5.339 s)
  Keyboard component
    √ CIM: renders the Keyboard component correctly (75 ms)
    √ CIM: fires onClick event for a key (20 ms)
    √ CIM: fires onDelete and onEnter events on keyboard events (16 ms)
```

Figure 3: Testing output of Keyboard.tsx



```
PASS  src/App.test.tsx (8.421 s)
  √ Integration: Clicking a key on the onscreen keyboard (105 ms)
  √ Integration: Clicking all 5 keys on the onscreen keyboard (210 ms)
  √ Integration: Clicking all 5 keys on the onscreen keyboard and validating response (94 ms)
  √ CIM: grid renders all inputted letters correctly  (13 ms)
  √ CIM: registers existing inputs appropriately and generates letters on screen (9 ms)
  √ Integration: renders App component (29 ms)
  √ CIM: renders Cell component without crashing (2 ms)
  √ CIM: renders Cell component correctly with default attributes (3 ms)
  √ CIM: renders Cell component correctly with absent status (2 ms)
  √ CIM: renders Cell component correctly with correct status (3 ms)
  getGuessStatuses
    √ CIM: guess statuses (2 ms)
```

Figure 4: Testing output of App.tsx